Generación de Triangulaciones de Delaunay Persistentes

Jainor Nestor Cardenas Choque

Orientador: Dr. Cristian López Del Alamo

Jurado:
Dr. Jorge Poco Medina (Universidad Católica San Pablo - Perú)
Dr. Manuel Loaiza Fernández (Universidad Católica San Pablo - Perú)
Dr. Erick Gómez Nieto (Universidad Católica San Pablo - Perú)
Dr. Alex Cuadros Vargas (Universidad Católica San Pablo - Perú)

Tesis presentada al
Centro de Investigación e Innovación en Ciencia de la Computación (RICS)
como parte de los requisitos para obtener el grado de
Maestro en Ciencia de la Computación.

Universidad Católica San Pablo – UCSP
Agosto de 2017 – Arequipa – Perú

Dedicado a Dios, a mi familia.

Agradecimientos

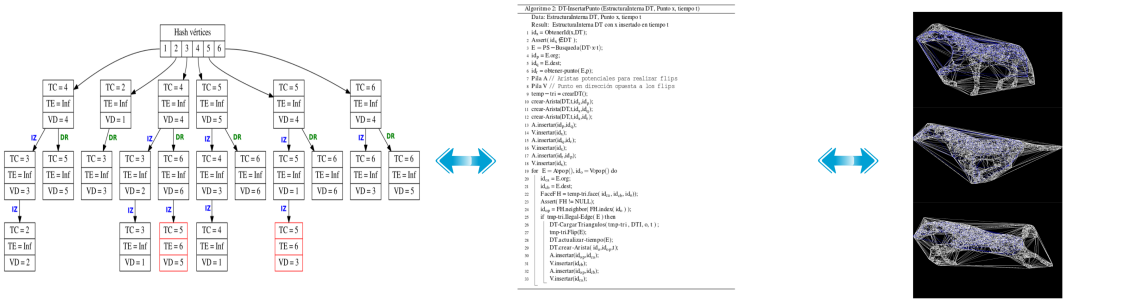
Agradezco a mis padres por el apoyo brindado para forjarme como profesional.

Agradezco a la universidad, mi *alma matter*, por haberme cobijado y brindado la formación que ahora me permitirá ayudar a construir una mejor sociedad.

Agradezco de forma muy especial a mi orientador Prof. Dr. Cristian López Del Alamo por haberme guiado en esta tesis.

Deseo agradecer de manera especial al Consejo Nacional de Ciencia, Tecnología e Innovación Tecnológica (CONCYTEC) y al Fondo Nacional de Desarrollo Científico, Tecnológico e Innovación Tecnológica (FONDECYT-CIENCIACTIVA), que mediante Convenio de Gestión UCSP-FONDECYT N 011-2013, han permitido la subvención y financiamiento de mis estudios de Maestría en Ciencia de la Computación en la Universidad Católica San Pablo (UCSP).

Abstract



Algorithms for generating Delaunay Triangulation are quite known and have many applications in various areas, such as computational geometry, astronomy, robotics, cartography, zoology, among others. This research proposes using concepts of space-time structures such as persistence, to design an algorithm for Delaunay Triangulation temporary space, so that it is feasible to perform queries and modifications in a given time t minimizing the spatial and temporal complexity.

Keywords: Persistent Data Structure, Delaunay Triangulation, Full Persistent.

Resumen

Los algoritmos para generar triangulaciones de Delaunay son ampliamente conocidos y tienen muchas aplicaciones en diversas áreas, como en geometría computacional, astronomía, robótica, cartografía, zoología, entre otros. La presente investigación, propone utilizar conceptos de estructuras espacio-temporales como la persistencia, para diseñar un algoritmo de triangulaciones de Delaunay, de modo que sea factible realizar consultas y modificaciones en un determinado tiempo t minimizando la complejidad espacial y temporal.

Palabras clave: Estructuras de datos persistentes, triangulación de Delaunay, Persistencia Completa.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Las estructuras de datos conforman un campo de estudio e investigación fundamental en ciencia de la computación, por este motivo la correcta elección y un adecuado diseño son indispensables para el desarrollo de algoritmos de almacenamiento y/o recuperación de información de manera eficiente. Sin embargo, la mayoría de estructuras suelen ser efímeras, lo que quiere decir que en cada operación de inserción, eliminación o modificación, éstas pierden los valores o estados anteriores (Böszörményi y Weich, 1996); por otro lado las estructuras de datos persistentes son conocidas como estructuras de datos temporales (Mehta y Sahni, 2004), éstas permiten mantener versiones anteriores de una estructura por lo que conservan estados anteriores cuando son modificados. Todo esto, teniendo en cuenta las características de costo computacional y espacial.

La triangulación de Delaunay es una estructura de datos fundamental en geometría computacional, computación gráfica, ingeniería y otras áreas de aplicación algorítmica (Fiat y Kaplan, 2001; Demaine et al., 2008). Por esta razón, su utilidad y la de su dual (*diagramas de Voronoi*), tienen diversas aplicaciones como por ejemplo: solución al problema de encontrar el punto más cercano (Shamos y Hoey, 1975), modelamiento de terrenos (Razafindrazaka, 2009), detección de colisiones (Aurenhammer, 1991), búsqueda asociativa de archivos, solución al problema de agrupamiento de datos, entre otros.

Por otro parte, la triangulación de Delaunay, que actualmente está implementada en librerías como CGAL, no cuentan con persistencia parcial ni persistencia total, lo cual implica que si se aplica un conjunto de operaciones como: inserción, eliminación o modificación sobre la estructura de datos en el tiempo, entonces, ésta pierde los estados anteriores. Esto ocurre, también, en todas las estructuras de datos tradicionales como: listas, pilas, colas, árboles binarios equilibrados, árboles B, B+ y muchas otras. Este tipo de estructuras tienen el problema de no poder regresar a un estado i anterior, después de aplicarles n modificaciones ($0 < i < n$); es por esta razón, que a este tipo de estructuras clásicas se les conoce como estructuras de datos efímeras.

Por ejemplo, si se modifica n veces el valor de un nodo en una lista simplemente enlazada, no es posible saber el valor que tenía en un tiempo i anterior ($0<i<n$). Esto se debe a que los cambios, en este tipo de estructuras, no son almacenados. Por otro lado, mantener todas

las versiones, en distintas copias, de una misma estructura de datos, es costoso. Entonces, surge el problema de desarrollar estructuras de datos que sean capaces de mantener distintas versiones de sí mismas, minimizando el costo de memoria y manteniendo el desempeño de las operaciones lo más cercanas posibles a la estructura original. A este tipo de estructuras de datos se les denomina estructuras de datos persistentes.

En general, es posible clasificar las estructuras de datos como parcialmente persistentes, completamente persistentes, confluentes y funcionales. Las primeras, son aquellas estructuras en las que solo es posible realizar modificaciones en el estado actual, sin embargo, se pueden realizar consultas en cualquier estado, es decir, en cualquier tiempo pasado; por otro parte, las estructuras de datos completamente persistentes son aquellas a las cuales se les puede aplicar modificaciones en cualquier tiempo pasado, y a la vez, es posible realizar consultas en cualquier tiempo o en cualquier estado. Luego, es claro que el diseño de estructuras de datos completamente persistentes es más complejo debido a la cantidad de memoria que pueden llegar a utilizar.

En la literatura actual, aún no existe una estructura de datos de una triangulación de Delaunay que sea completamente persistente; en tal sentido, esta tesis tiene un aporte importante: la propuesta de una nueva estructura de datos persistente para representar una triangulación de Delaunay.

1.1. Justificación

Existen muchas librerías para el cálculo de la triangulación de Delaunay, como por ejemplo CGAL, Fade2D, entre otros. Estas librerías tienen algoritmos eficientes para el cálculo de las triangulaciones de Delaunay; sin embargo, no poseen características de persistencia; es decir, no cuentan con la posibilidad de mantener versiones de las estructuras cada vez que se realizan operaciones sobre ellas.

Además, un aspecto importante que deberían tener las estructuras de datos, es la capacidad de mantener versiones pasadas de ellas mismas, con el fin de regresar a una versión t anterior; de modo que sea factible evaluar las características de la estructura en ese tiempo.

En ese sentido, esta investigación propone el desarrollo una estructura de datos persistente para el manejo de una triangulación de Delaunay. Esta estructura de datos, además de mantener las versiones pasadas de la estructura, mantendrá la complejidad computacional y minimizará la complejidad espacial.

1.2. Motivación

Como se mencionó anteriormente, además de la falta de una estructura de datos persistente para la triangulación de Delaunay, la existencia de tal estructura de datos sería útil en diferentes aplicaciones, como por ejemplo, en un mapa bidimensional en el que se han

marcado diferentes puntos, los cuales indican regiones donde se han originado diferentes fenómenos naturales. En este modelo se desean realizar simulaciones acerca de las posibles zonas cercanas que pudieran verse afectadas. En tal sentido, la triangulación persistente de Delaunay resulta una estructura de datos natural para este problema, puesto que esta busca minimizar consumo de memoria RAM en aquellos problemas computacionales que puedan ser representados o resueltos utilizando triangulaciones de delaunay variables en el tiempo producto de inserciones, eliminaciones y/o modificaciones.

Por otro lado, imaginemos que tenemos una base de datos de imágenes satelitales, de un río, tomadas en distintos periodos de tiempo. Podemos representar la imagen del río mediante una triangulacion de Delaunay. Sin embargo, con el paso del tiempo es claro que las características del cauce del río cambien, aumentando caudal o modificando su forma, esto hace que se tengan varias imágenes del mismo río en diferentes fechas; por lo tanto, se deberían de construir diferentes triangulaciones de Delaunay. Sin embargo, nuestro método genera una única triangulación de Delaunay, el cual permite minimizar la memoria RAM y mantener la performance de reconstrucción de la triangulación en un tiempo específico.

1.3. Planteamiento del problema

Las estructuras de datos efímeras no poseen mecanismos para poder realizar consultas en tiempos pasados, en tal sentido, una de las formas, más triviales, de mantener la información de las versiones generadas por las modificaciones en las estructuras de datos, es mediante copias de sí mismas luego de cada modificación. Este método es, claramente, contraproducente en cuanto a gasto de memoria se refiere.

Por otro lado, las estructuras de datos que actualmente son conocidas y utilizadas para representar la triangulación de Delaunay son efímeras, es decir, no es factible realizar operaciones o consultas sobre los datos históricos de la triangulación en un determinado tiempo t, ni operaciones para realizar cambios en la estructura en un tiempo pasado. Sin embargo, en la actualidad existen métodos para mantener versiones de algunas estructuras de datos triviales como listas, pilas, colas y árboles (Demaine et al., 2008). Estos métodos, tienen la capacidad de mantener la información de las versiones pasadas minimizando la cantidad de memoria RAM y manteniendo la complejidad algorítmica de sus operaciones.

En tal sentido, en esta tesis, se plantea e implementa una nueva estructura de datos parcialmente persistente y completamente persistente, que pueda realizar consultas y ejecutar operaciones (en cualquier tiempo anterior t) minimizando costo computacional y uso de memoria RAM.

1.4. Objetivo

Proponer e implementar una estructura de datos persistente de la triangulación de Delaunay, de modo que sea posible, realizar consultas y modificaciones en versiones anteriores, minimizando la complejidad computacional y la complejidad espacial.

1.5. Aportes

Los aportes de esta investigación son:

- Se plantea un nuevo algoritmo para la generación de la triangulación de Delaunay persistente, manteniendo la performance en memoria y la complejidad algorítmica en cada operación.
- Se implementa y se analiza una versión parcialmente persistente y una totalmente persistente de la triangulación de Delaunay.

Capítulo 2

Conceptos preliminares y trabajos previos

En el siguiente capítulo se explican conceptos y trabajos relacionados de las estructuras de datos persistentes. En primer lugar, se describirán los tipos de persistencia y los mecanismos desarrollados para mantener persistencia en algunas estructuras de datos básicas. En segundo lugar, se explicarán los trabajos previamente desarrollados, según el estado del arte, para mantener algunas estructuras de datos persistentes.

2.1. Estructuras de datos persistentes

Una estructura de datos es persistente si tiene la capacidad de mantener versiones de sí misma producto de operaciones como inserción, eliminación o modificaciones, que cambian la estructura en el tiempo (Mehta y Sahni, 2004).

Por ejemplo, en el caso de un árbol binario, al realizar una operación sobre la estructura, ésta incrementa, decrementa, o cambia las claves de los nodos según el tipo de operación: inserción, eliminación o modificación. Sin embargo, para poder saber el estado de la estructura en un tiempo t pasado, sería necesario almacenar cada versión antes de una modificación, la manera más simple sería guardar en memoria RAM toda la estructura de datos antes de sufrir una modificación; de esta forma, se tendrían n copias distintas de la estructura, una para cada modificación en el tiempo, de tal manera, que recuperar el estado de la estructura en un tiempo t, implicaría instanciar una copia r, en ese tiempo t. No obstante, esto implicaría sacrificar espacio de memoria.

En este sentido, las estructuras de datos persistentes, buscan desarrollar estrategias para minimizar el espacio requerido por las nuevas versiones que se generan a través del tiempo.

2.2. Tipos de Persistencia

2.2.1. Persistencia Parcial

Una estructura de datos es parcialmente persistente, sí permite hacer modificaciones sólo en la estructura de datos actual, pero es posible hacer consultas de cualquier versión anterior y en cualquier tiempo pasado t, donde $0 < t <= n$, y n es el tiempo de creación de la última versión (Demaine et al., 2007).

Estas versiones pueden ser accedidas, debido a que cada nodo almacena información adicional. Esta información extra se refiere al tiempo de creación de esa versión, lo cual indica, el momento en que se hizo la modificación. El tiempo, de cada versión, es conocido como *timestamp*. Un ejemplo del comportamiento de esta estructura puede verse en la Figura 2.2.1 (a), donde cada nodo representa una versión de la estructura de datos.

2.2.2. Persistencia Completa

Una estructura de datos es completamente persistente sí permite hacer tanto consultas como modificaciones en cualquiera de las versiones de la estructura de datos. Con este tipo de persistencia, las versiones no forman una trayectoria lineal simple, en su defecto, éstas forman un árbol de versiones (Demaine et al., 2007). Esto se puede ver en la Figura 2.2.1 (b).

2.2.3. Confluente o funcional

Este tipo de persistencia, además de permitir modificaciones de versiones pasadas en el tiempo y realizar consultas en el pasado; también permite unir dos o más versiones anteriores para crear una nueva versión. Esto hace que las versiones tomen la forma de un DAG (*direct acyclic graph*), lo cual se puede observar en la Figura 2.2.1 (c).

En la Figura 2.2.1 se puede observar los diferentes tipos de persistencia. En (a) se aprecia la persistencia parcial. Los nodos grises son versiones anteriores, mientras que el nodo de color azul es el único que se puede modificar; manteniendo, de esta forma, una estructura lineal de persistencia. En (b) se aprecia una estructura de datos completamente persistente. Esta estructura puede modificar cualquiera de las versiones en el tiempo y, por lo tanto, las versiones conforman un árbol, de modo que cualquier versión en el tiempo puede ser modificada y consultada.

Teniendo en cuenta que la manera obvia de proporcionar persistencia es hacer una copia de la estructura de datos cada vez que se realiza una operación de modificación en la estructura y, que utilizar este método tendría el inconveniente de requerir espacio y tiempo proporcional al espacio ocupado por la estructura de datos original; es que se requiere un método para minimizar la complejidad espacial y temporal.

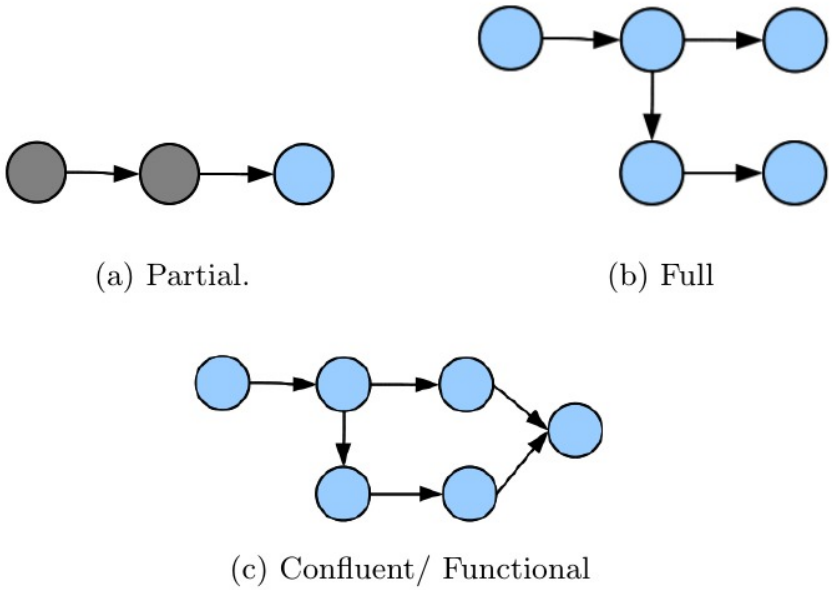


Figura 2.2.1: Tipos de Persistencia (Demaine et al., 2007).

Se llama a una estructura de datos persistente si es compatible con el acceso a múltiples versiones. La estructura es parcialmente persistente si se puede acceder a todas las versiones, pero sólo la versión más reciente se puede modificar, y se denomina totalmente persistente si cada versión se puede acceder y/o modificar (Driscoll et al., 1986).

Por otro lado, se han ideado formas persistentes de varias estructuras de datos, incluyendo pilas, colas, búsqueda en árboles, etc. La mayoría de estos resultados utilizan construcciones *Ad hoc*; con la excepción del artículo de Overmars (1981) y de Driscoll et al. (1986).

2.3. Métodos para crear una estructura de datos persistente

En la literatura se han planteado diversos métodos para hacer una estructura de datos persistente (Driscoll et al., 1986), a continuación, se muestran esquemas generales para convertir a una estructura de datos efímera en una estructura de datos persistente.

2.3.1. *Fat Nodes*

En este esquema, se registran todos los cambios realizados en los campos del nodo sin borrar los antiguos valores de los campos. Esto hace que los nodos de la estructura se deno-

minen *Fat Nodes*, donde cada uno contiene los mismos campos de información y punteros que un nodo efímero, pero mantienen espacio extra para agregar valores adicionales como los datos antiguos y los tiempos (*timestamp*) en los cuales se realizaron las modificaciones (Mehta y Sahni, 2004).

Cada dato adicional tiene un valor asociado y un *timestamp* que indica la versión en la que el campo fue cambiado por un valor en específico, además cada *Fat Node* tiene su propio *timestamp*, lo que señala la versión en la que se creó el nodo.

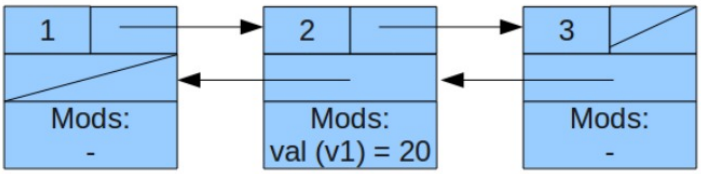


Figura 2.3.1: Ejemplo de *Fat Node* de una lista simplemente enlazada (Demaine et al., 2008).

En la Figura 2.3.1 se aprecian *Fat Nodes* en una lista simplemente enlazada. Cada nodo tiene, además de sus datos y punteros, un campo adicional el cual almacena nuevas versiones.

Por otro lado, debido a que los *Fat Nodes* sólo tienen espacio adicional limitado a un número fijo de modificaciones; entonces, en el caso que un nodo tenga todos sus campos de información completamente llenos, es necesario crear un nuevo nodo. Este nuevo nodo debe de mantener punteros al nodo anterior y siguiente en la lista; además, debe ser apuntado por los nodos anterior y siguiente, sin perder la dirección del nodo de versión anterior. Esto se puede observar en la Figura 2.3.2

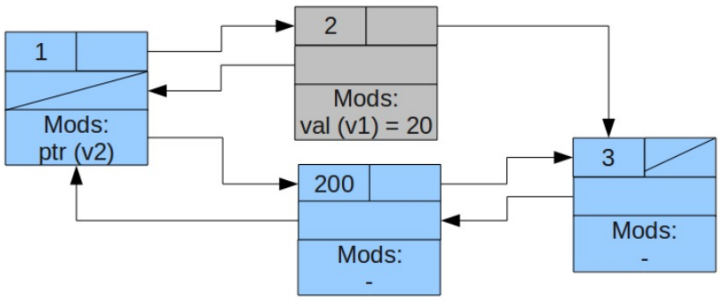


Figura 2.3.2: Ejemplo de *Fat Node* de una lista simplemente enlazada cuando un nodo tiene su espacio adicional saturado (Demaine et al., 2008).

2.3.2. *Path Copying*

Path Copying es un método que consiste en hacer una copia, de todos los nodos de la ruta o camino, que contiene el nodo que se está por modificar, esta operación se realiza

en cascada para modificar todos esos nodos, de modo que todos los nodos que apuntaban al antiguo nodo ahora deben ser modificados para apuntar al nuevo nodo.

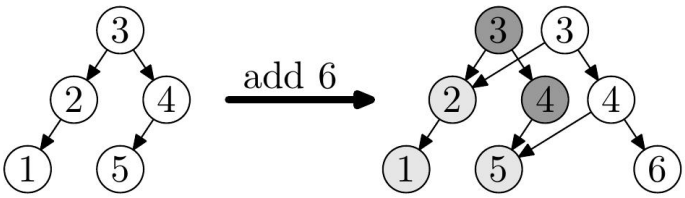


Figura 2.3.3: *Path Copying* en un árbol binario de búsqueda. La versión antigua (v1) consiste de nodos negros y grises; la nueva versión (v2) consiste de nodos blancos y nodos grises (Demaine et al., 2008).

Estas modificaciones generan cambios en cascada, hasta llegar a la raíz del árbol; lo cual hace que cada versión tenga su propia raíz, y en tal sentido, se debe mantener una serie de raíces indexadas por tiempo como se muestra en la Figura 2.3.3. La estructura de datos a la que apunta la raíz del tiempo t es exactamente la estructura del tiempo t (Mehta y Sahni, 2004).

Existen maneras más complejas de hacer persistencia, como persistencia confluente (Fiat y Kaplan, 2001; Kaplan et al., 2000; Buchsbaum y Tarjan, 1995). Esta estructura permite que dos versiones de una estructura de datos se junten en una sola operación de actualización. Esta investigación se centra en persistencia parcial y completa, por lo que se limita a estos dos tipos fundamentales de persistencia.

2.4. Triangulación de Delaunay

En esta sección se describe que es un triangulación de Delaunay, así como su importancia.

2.4.1. Triangulación de un conjunto de vértices en el plano

A partir de un conjunto $P := \{p_1, p_2, ..., p_n\}$ conformado por n vértices en el plano, se define una triangulación T de P como una subdivisión máxima planar donde ninguna arista que conecta dos vértices puede ser agregada sin cambiar su planaridad (De Berg et al., 2000), además siempre existe una triangulación de P, dado que cualquier polígono puede ser triangulado (Mirzaian, 1988), por otra parte las aristas de la envolvente convexa (Ramaswami, 1993) de P pertenecen a cualquier triangulación T que éste conformada por P y que la cara sin límites o infinita es siempre complemento de la envolvente convexa.

Por otro lado, el número de triángulos es el mismo en cualquier triangulación de P, así también como el número de aristas, además estos dependen del número de vértices en P que se encuentran en el límite de la envolvente convexa de P (incluyendo los vértices que se encuentren en alguna arista de la envolvente convexa), como muestra el teorema 2.1.

Teorema 2.1. *Sea P un conjunto de n vértices en el plano, no todos colineales, y k denota el número de vértices en P que se encuentran en el límite del convex hull de P. Entonces, cualquier triangulación de P tiene $2n-2-k$ triángulos y $3n-3-k$ aristas (De Berg et al., 2000).*

Si se considera m como el número de triángulos que conforman T, cualquier otra triangulación de P también estará conformada por m triángulos, considerando los $3m$ ángulos de los triángulos de T, ordenados por su valor en orden creciente, como $\alpha_1, \alpha_2, ...\alpha_{3m}$, por lo tanto, $\alpha_i \leq \alpha_j$ para $i < j$, se denomina $A(T) = (\alpha_1, \alpha_2, ...\alpha_{3m})$ el ángulo-vector de T.

Sea T' otra triangulación del mismo conjunto de vértices P, y sea $A(T') = (\alpha'_1, \alpha'_2, ...\alpha'_{3m})$ su ángulo-vector, el ángulo-vector de T es mayor que el ángulo-vector de T' si $A(T)$ es lexicográficamente mayor que $A(T')$, es decir existe un índice $1 \leq i \leq 3m$ tal que $\alpha_j = \alpha'_j$ para todo $j < i$ y $\alpha_i > \alpha'_i$, lo cual se denota como $A(T) > A(T')$, además se puede llamar a una triangulación T ángulo-óptima si $A(T) \geq A(T')$ para todas las triangulaciones T' de P.

Para ver si una triangulación es ángulo-óptimo se utiliza el teorema 2.2, llamado Teorema de Thales, donde se denota el menor ángulo definido por tres vértices p, q, r por $\measuredangle pqr$.

Teorema 2.2. *Sea C un círculo, l una línea intersecando C en los vértices a y b, sea p, q, r y s vértices que se encuentran en el mismo lado de l. Se supone que p y q se encuentran en C, además que r se encuentra dentro de C, y que s está fuera de C. Entonces:*

$$\measuredangle arb > \measuredangle apb = \measuredangle aqb > \measuredangle asb \tag{2.1}$$

A partir del teorema 2.2, se considera una arista $e = p_i p_j$ de una triangulación T de P que no es una arista que se encuentra en la envolvente convexa, es decir, es incidente a dos triángulos $p_i p_j p_k$ y $p_i p_j p_l$. Si estos dos triángulos forman un cuadrilátero convexo, podemos obtener una nueva triangulación T' mediante la eliminación de $p_i p_j$ de T y la inserción de $p_k p_l$ en su lugar. A esta operación se le llame *flip edge* .

La única diferencia en el ángulo-vector de T y T' son los seis ángulos $(\alpha_1, \alpha_2, ...\alpha_6)$ en $A(T)$, los cuales son reemplazados por $(\alpha'_1, \alpha'_2, ...\alpha'_6)$ en $A(T')$. La Figura 2.4.1 muestra este comportamiento.

Por otro lado, se llama a dicha arista e ilegal si se cumple la inecuación 2.2, es decir si al realizar la operación flip se puede aumentar el valor del menor ángulo.

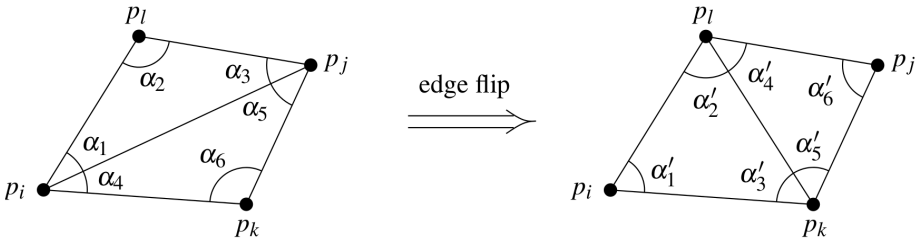


Figura 2.4.1: Operación *flip* (De Berg et al., 2000).

$$\min_{1\leq i\leq 6} \alpha_i < \min_{1\leq i\leq 6} \alpha'_i \tag{2.2}$$

De otro modo, se puede tomar en cuenta el siguiente lema para poder corroborar si una arista es ilegal.

Lema: Sea la arista $p_i p_j$ incidente a los triángulos p_i, p_j, p_k y p_i, p_j, p_l y sea C el círculo que pasa por p_i, p_j, p_k; entonces la arista $p_i p_j$ es ilegal si y sólo si el vértice p_l se encuentra en el interior de C. Por otra parte, si los vértices p_i, p_j, p_k, p_l forman un cuadrilátero convexo y no están en un círculo común, entonces $p_i p_j$ o $p_k p_l$ es una arista ilegal.

Además cuando los cuatro vértices se encuentran en un círculo, tanto $p_i p_j$ y $p_k p_l$ son legales. Por otro lado los dos triángulos incidentes a una arista ilegal debe formar un cuadrilátero convexo, por lo que; siempre es posible hacer *flip* a una arista ilegal.

En la teoría, se define una triangulación legal como una triangulación que no contiene alguna arista ilegal, también de la inecuación 2.2 se concluye que cualquier triangulación ángulo-óptima es legal. Además una triangulación de Delaunay es siempre legal como se muestra en la definición 2.3.

Teorema 2.3. *Sea P un conjunto de vértices en el plano. Una triangulación T de P es legal, si y sólo si, T es una triangulación de Delaunay de P.*

Dado que cualquier triangulación ángulo-óptimo debe ser legal, el teorema 2.3 implica que cualquier triangulación ángulo-óptima de P es una triangulación de Delaunay de P.

Cuando P está en posición general, sólo hay una triangulación legal, que es entonces la única triangulación ángulo-óptimo, es decir, la única triangulación de Delaunay. Cuando P no está en posición general, existen muchas triangulaciones legales de P, no todas estas triangulaciones de Delaunay serán de ángulo óptimo, sin embargo todas estas triangulaciones tendrán el mismo valor para el ángulo mínimo de sus ángulo-vectores, esto se puede observar en el teorema 2.4 y ser verificado por el teorema de Tales.

Teorema 2.4. *Sea P un conjunto de vértices en el plano. Cualquier triangulación de ángulo-óptimo de P es una triangulación de Delaunay de P. Además, cualquier triangulación de Delaunay de P maximiza el ángulo mínimo sobre todas las triangulaciones de P.*

De lo visto anteriormente, para obtener una triangulación de Delaunay T a partir de una triangulación cualquiera T' ambas triangulaciones del conjunto de vértices P, basta verificar si todas las aristas son legales y en caso de no serlo, realizar operaciones *flips*, sobre las aristas ilegales, hasta que no existan aristas ilegales en T'.

2.4.2. *Constrained Delaunay Triangulation*

En algunos casos, es necesario forzar la creación de ciertas aristas dentro de una triangulación, estas son llamadas *Constrained Delaunay Triangulations* (Chew, 1989).

2.4.3. Algoritmos para construir la triangulación de Delaunay

La dualidad entre la triangulación de Delaunay y diagramas de Voronoi es extensamente conocida (Fortune, 1992), (Preparata y Shamos, 2012) y, por lo tanto, los algoritmos para la construcción de diagramas de Voronoi sirven para calcular la triangulación de Delaunay. Sin embargo, los métodos directos de construcción son generalmente más eficientes debido a que el diagrama de Voronoi no tiene que ser calculado y almacenado.

Los algoritmos directos para calcular triangulaciones de Delaunay (Aurenhammer, 1991) pueden ser clasificados de la siguiente manera (Cignoni et al., 1998):

- Mejoras locales: A partir de una triangulación arbitraria, estos algoritmos modifican localmente las aristas de los pares de vértices adyacentes a través de operaciones *flips* hasta obtener una triangulación de Delaunay.
- Inserción incremental: Estos algoritmos insertan los vértices uno a la vez. El triángulo que contiene el nuevo vértice se divide insertando nuevos triángulos adyacentes al nuevo vértice; luego, el criterio del circuncírculo es probado en todas los vértices adyacentes y, si es necesario, se aplica la operación *flip* a las aristas.
- Construcción gradual: Estos algoritmos transforman los vértices en el espacio de d a $d+1$ dimensiones y luego calculan la envoltura convexa de los vértices transformados, la triangulación de Delaunay se obtiene con la proyección de la envoltura convexa en d dimensiones (Avis y Bremner, 1995).
- Divide y vencerás: Se basan en la partición recursiva y la triangulación local de un conjunto de vértices, para luego realizar una fase de combinación, donde se unen las triangulaciones resultantes (Lee y Schachter, 1980).

2.5. Algoritmo Incremental

Se calcula la triangulación de Delaunay de un conjunto de vértices P mediante la inserción de un vértice tras otro, es decir, se mantiene la triangulación del conjunto de vértices P insertados hasta el momento R, donde $R \subseteq P$. Para la inserción del siguiente vértice s, se actualiza la triangulación para obtener la triangulación de Delaunay de $R \cup \{s\}$.

Para evitar casos especiales, se agrega al conjunto de vértices P tres vértices artificiales (un triángulo) que deben de envolver a todos los vértices, es decir, la envolvente convexa del conjunto de vértices resultante es un triángulo. El algoritmo incremental comienza con la triangulación de Delaunay de los tres vértices artificiales.

En cada inserción de un vértice s, se debe de encontrar el triángulo $\Delta = \triangle(p,q,r)$ de $DT(R)$ que contiene s, y sustituirlo por los tres triángulos resultantes de s conectado con los tres vértices p,q,r, como se observa en la Figura 2.5.1.

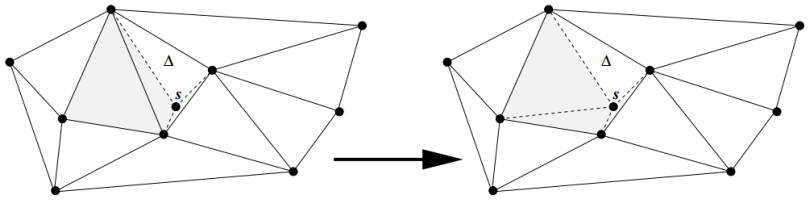


Figura 2.5.1: Inserción incremental (Bernd Gärtner, 2013).

Para encontrar el triángulo que contiene el vértice a insertar, se plantean diferentes algoritmos y/o estructuras de datos:

- Estructuras de datos para consultas del vecino más cercano (Roussopoulos et al., 1995).
- Algoritmos para caminar en triangulaciones (Brown y Faigle, 1997).

Dada la triangulación T de $R \cup \{s\}$, se realizan *flips* en T hasta obtener DT($R \cup \{s\}$) se obtiene como en la Figura 2.5.2.

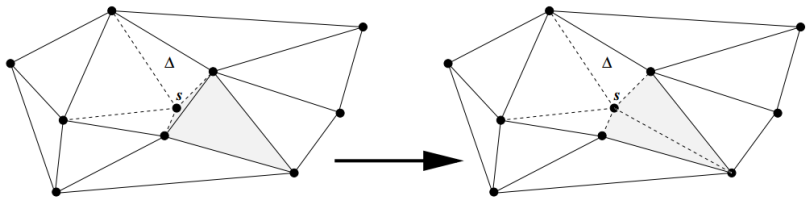


Figura 2.5.2: Resultado inserción incremental (Bernd Gärtner, 2013).

Cada arista incidente de s que se crea durante la actualización es una arista del grafo de Delaunay de $R \cup \{s\}$ y por lo tanto una arista que estará en $DT(R \cup \{s\})$.

2.6. Trabajos Relacionados

El uso de estructuras de datos persistentes está involucrado en muchas áreas de la ciencia de la computación tales como programación funcional, geometría computacional y otras áreas de aplicación de algoritmos (Mehta y Sahni, 2004), la mayoría de estas aplicaciones requieren poder realizar consultas en estados anteriores de la estructura de datos.

Existe un esquema ingenuo para hacer cualquier estructura de datos persistente, en este esquema se llevan a cabo las operaciones tal y como se han llevado de manera efímera pero antes de que cada operación de actualización se realice en la versión actual, se crea nuevas copias sobre las cuales se realizará la operación requerida, manteniendo en memoria ambas versiones independientemente una de la otra, esto es obviamente ineficiente dado que el consumo de tiempo y espacio es igual al número de elementos de cada versión.

De manera general, las estructuras de datos persistentes se dividen en dos tipos: La primera, vista en el capítulo anterior, intenta crear maneras generales que harían que cualquier estructura de datos efímera sea persistente, mientras se mantiene bajo costo computacional y espacial; la segunda, la cual presenta soluciones específicas para generar estructuras de datos persistentes, algunas de ellas se describen a continuación.

2.6.1. Ubicación de un vértice en el plano

"Una de las aplicaciones geométrica más conocida es el algoritmo para la ubicación de un vértice en el plano que desencadenó el desarrollo de toda el área. En el problema de localización de un vértice en un plano se nos da la subdivisión del plano euclidiano en polígonos de n segmentos de líneas que se cruzan solamente en sus extremos" (Sarnak y Tarjan, 1986).

Primero se debe hacer un procesamiento sobre los segmentos de línea, de tal modo que dado un vértice de consulta p se pueda determinar de manera eficiente cuales son los polígonos que lo contienen. Se divide el plano en franjas verticales trazando una línea vertical que pasa por cada vértice, es decir las intersecciones de los segmentos de línea en la subdivisión del plano (Mehta y Sahni, 2004), la intersección de los segmentos de línea con subdivisión de un fragmento se encuentran ordenados, entonces es posible responder a cada consulta mediante dos búsquedas binarias. Una búsqueda binaria localiza el fragmento que contiene la consulta, y otra búsqueda binaria localiza el segmento anterior al vértice de consulta dentro del fragmento.

Si tenemos el conjunto de un segmento en concreto se puede obtener el conjunto de segmentos mediante la eliminación de los segmentos que terminan en el límite entre éstos, y la inserción de segmentos que empiezan en ese límite. A medida que se barre todas los segmentos de izquierda a derecha tenemos que en total hay n eliminaciones y n inserciones; una eliminación y una inserción por cada segmento de línea, con lo cual el problema se reduce a mantener árboles de búsqueda parcialmente persistentes para realizar las consultas.

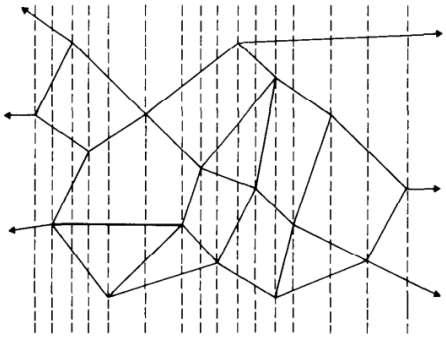


Figura 2.6.1: Subdivisión del plano (Sarnak y Tarjan, 1986).

2.6.2. Array Persistentes

En el artículo de Dietz (1989) se muestra una técnica general para la creación de arreglos persistentes, la cual necesita tiempo $O(loglog(m))$ para acceder al arreglo y $O(loglog(m))$ amortizado (Tarjan, 1985) para cambiar el contenido de una entrada, donde m es el número total de cambios siendo el tamaño de espacio utilizado m.

De otro modo, n indica el tamaño del arreglo y se supone que $n < m$, se considera el array como un *Fat Node* con n campos. La lista de valores-versiones que describen las asignaciones a cada entrada del arreglo está representado en una estructura de datos propuesto por van Emde Boas et al. (1976), esta estructura de datos tiene complejidad espacial lineal al número de elementos, haciendo uso de hashing perfecto dinámico (Dietzfelbinger et al., 1994).

2.6.3. Triangulaciones Persistentes

En Blelloch et al. (2001) se describe un método para construir *Convex Hull* en 3 dimensiones, para posteriormente obtener la triangulación de Delaunay a partir de las aristas de la envolvente convexa, para obtener la Triangulación de Delaunay parcialmente persistente se mantiene información acerca de las aristas, las cuales son almacenadas en estructuras de datos con un tiempo de consulta óptimo, en particular, haciendo uso de árboles balanceados de búsqueda, para poder realizar consultas en la estructura de datos en un tiempo anterior t.

Muchos algoritmos, incluyendo el propuesto por Blelloch et al. (2001), se basan en la construcción de la cápsula convexa en la dimensión $d+1$ donde d es la dimensión. Es decir, dado un vértice p exterior a la cápsula convexa de un conjunto de vértices, se puede extender dicha cápsula para incluir esté vértice de la siguiente manera, considerar el vértice p exterior como una fuente de luz que ilumina un subconjunto de las caras de la cápsula. Estas caras iluminadas son eliminadas en la construcción.

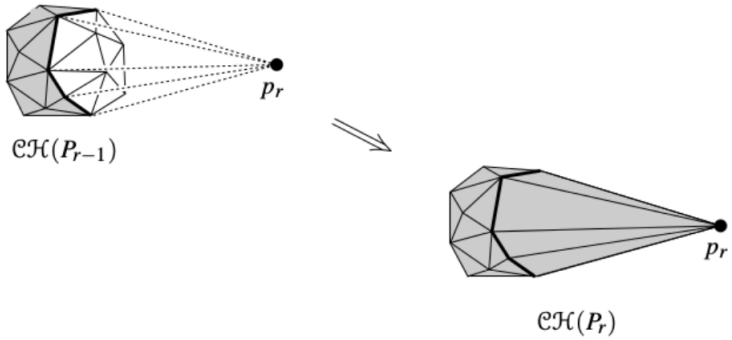


Figura 2.6.2: Algoritmo incremental, cápsula convexa en 3 dimensiones (Seidel, 1986).

El límite de las caras iluminadas es un conjunto de aristas denominadas el horizonte, a continuación se crea un poliedro piramidal como en la Figura 2.6.2, cuyo vértice es el vértice exterior y cuya base es el horizonte, esta construcción el vértice p como un nuevo vértice de la componente conexa, los vértices interiores a la cápsula, se descartan.

De otro modo, para obtener la Triangulación de Delaunay parcialmente persistente, se mantiene información acerca de las adyacencias, las cuales son almacenadas en estructuras de datos con un tiempo de consulta óptimo, en particular uso de árboles balanceados de búsqueda.

Capítulo 3

Triangulación de Delaunay Persistente

En el presente capítulo se explican los algoritmos planteados para la implementación de la triangulación de Delaunay persistente.

Se propone 3 algoritmos *Walk*, *Híbrido* y *Graph* para la operación de inserción en persistencia parcial, también se propone la operación de eliminación de un vértice para el método *Walk*. Posteriormente se extiende la propuesta Walk para la triangulación de Delaunay persistente completa.

3.1. Método *Walk*

La propuesta consta de una estructura, la cual se divide en dos partes fundamentales:

- La primera, corresponde a las estructuras de datos, las cuales darán soporte a la triangulación de Delaunay persistente y a cuya etapa denominaremos **estructura interna.**
- La segunda, describe la **inserción y eliminación persistente** que dará soporte a la triangulación de Delaunay propuesta.

3.1.1. Estructura interna

La estructura interna, a diferencia de librerías como CGAL, considera persistencia de versiones anteriores; es decir, permite almacenar estados anteriores, las cuales fueron generadas producto de alguna operación de modificación. El costo de almacenamiento de versiones anteriores en la estructura de datos interna minimizará el costo temporal y mantendrá la complejidad computacional.

Una de las formas, más simples, pero menos eficientes de mantener las versiones sería mediante el almacenamiento de copias completas de todas las versiones pasadas; y teniendo

en cuenta que cada vez que se realiza una operación de inserción o eliminación, se genera un nuevo estado o versión, entonces; implementar este método sería contraproducente en tiempo y en espacio.

En tal sentido, la estructura interna, tiene como objetivo minimizar la cantidad de memoria necesaria para mantener las versiones de la estructura y, además, mantener la complejidad algorítmica en cada una de las operaciones.

Teniendo en cuenta que cada operación de inserción o eliminación, en la triangulación de Delaunay, modifica las aristas de uno o varios vértices; entonces, se propone utilizar una estructura de datos adicional, la cual permitirá almacenar los cambios, solo de las aristas y de los vértices afectados en un determinado tiempo t.

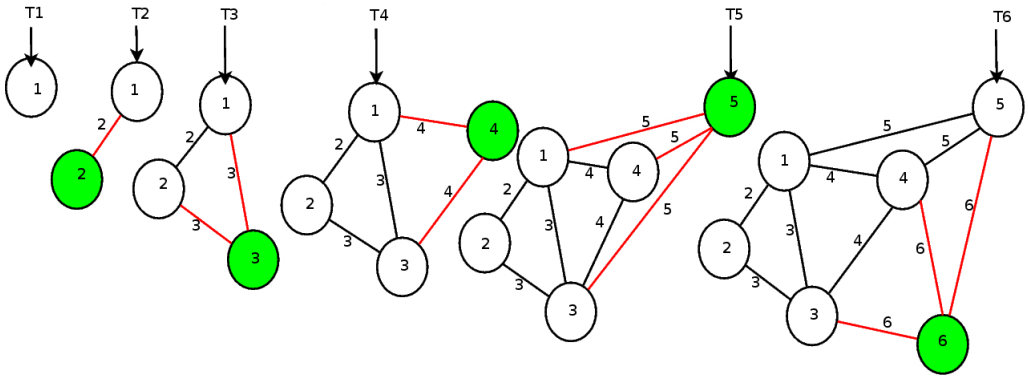


Figura 3.1.1: Triangulación a través del tiempo. Donde T_1 representa la triangulación de Delaunay inicial y para las demás triangulaciones, T_i representa la triangulación de Delaunay de $T_{i-1} \cup$ { vértice i }.

En la Figura 3.1.1, se puede observar los cambios de la triangulación a través del tiempo luego de realizar operaciones de inserción. Los nodos de color blanco, representan vértices que fueron ingresados en un tiempo pasado y, por otro lado, los nodos de color verde son aquellos que están siendo ingresados en el tiempo T_i. Cada nodo almacena información del tiempo T_i en el que fue insertado, las aristas almacenan información del tiempo en que fueron creadas , de similar manera las aristas en rojo representan las aristas creadas en el tiempo T_i.

La estructura interna de la triangulación de Delaunay persistente, está conformada por un conjunto de estructuras de datos, las cuales permiten mantener la información de los cambios que se realizan sobre la triangulación. Estas estructuras adicionales, tienen como objetivo principal minimizar la cantidad de memoria RAM, lo cual va a permitir almacenar cualquier modificación y, realizar consultas en el pasado de la estructura, manteniendo el costo computacional de la regeneración de la triangulación en un tiempo t anterior.

Por otro lado, se plantea almacenar información sobre la vecindad de cada vértice a través del tiempo (vértices que en algún momento tuvieron alguna arista en común). Un ejemplo se observa en la Figura 3.1.1, donde cada arista tiene asociada un número entero, el cual indica el tiempo en el que fue creada. Además, cada vértice almacena un conjunto

de datos, los cuales representan la información histórica sobre vértices con los que se tuvo alguna relación (arista) en el pasado.

En la Figura 3.1.2 podemos observar la representación persistente de la triangulación de Delaunay de la Figura 3.1.1. En la Figura, se muestra una tabla *hash* que indexa los vértices de la triangulación de Delaunay; cada vértice tiene asociado un árbol binario de búsqueda balanceado y, además cada nodo del árbol simboliza una arista entre el vértice p de la tabla hash y un vértice destino q en T; incluso, mantiene información del tiempo de creación (TC) y el tiempo de eliminación(TE) de la arista entre p *y* q. Por ejemplo, el primer árbol indica que el vértice 1 está conectado con los vértices (VD) $2,3,4$ *y* 5 y la raíz de este árbol, es decir, el nodo $[TC = 4, TE = Inf, VD = 4]$, indica que la arista fue creada en el tiempo 4, que no ha sido eliminada y que está conecta con el vértice 4 (respectivamente).

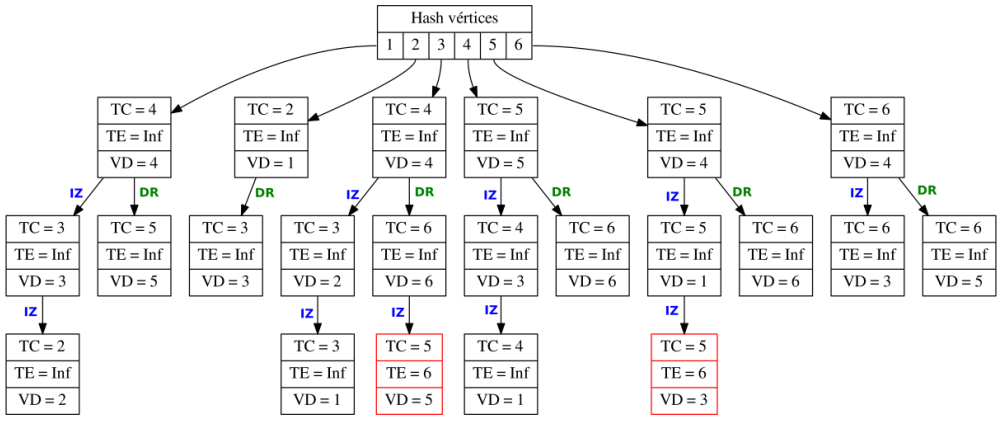


Figura 3.1.2: Representación Interna, cada vértice en la triangulación está asociado a un árbol binario de búsqueda balanceado.

En otras palabras, cada nodo del árbol, tiene la estructura $[TC, TE, VD]$, donde TC representa el tiempo de creación de la arista, TE es el tiempo de eliminación de la arista, el valor Inf representa el valor infinito, lo cual significa que la arista no ha sido eliminada, y finalmente VD, es el vértice adyacente. Un ejemplo de una arista que fue eliminada en el tiempo 6 se puede observar, en esta misma Figura, en los nodos de color rojo.

3.1.2. Operación de Inserción

En la presente investigación, la triangulación se representa como un conjunto de vértices y caras. Cada cara o triángulo proporciona acceso a sus tres vértices y sus tres caras vecinas en sentido horario *cw* o antihorario *ccw*. Cada cara tiene tres punteros a sus tres vértices y tres punteros a las tres caras adyacentes.

Estos punteros son indexados por 0, 1 y 2 en sentido antihorario, de tal manera que, en cada cara, el vértice indexado por i es opuesta a la cara adyacente con el mismo índice; como se muestra en la Figura 3.1.3 (Boissonnat et al., 2000).

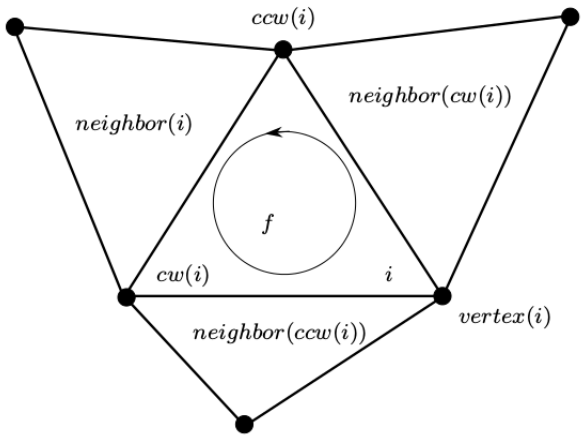


Figura 3.1.3: Representación de una triangulación (Boissonnat et al., 2000).

Para la operación de inserción utilizaremos el algoritmo incremental visto en la sección 2.5. Este método es utilizado debido a que la persistencia en el tiempo requiere almacenar información cada vez que se realiza una operación. Es decir, no es factible insertar n vértices en conjunto, como lo haría el método divide y vencerás.

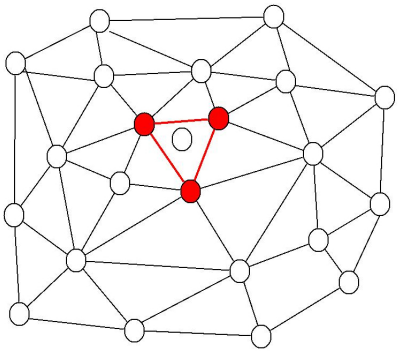


Figura 3.1.4: Ubicación del triángulo.

Tal como se plantea en el algoritmo incremental, el primer paso para poder realizar la inserción en la triangulación de Delaunay, es encontrar el triángulo que contiene el vértice a insertar. Ver Figura 3.1.4. Para encontrar este triángulo se implementó una modificación al algoritmo propuesto por Brown y Faigle (1997) cuyo costo computacional es $O(\sqrt{(n)})$; esto debido a que el algoritmo tenía que ser adaptado para funcionar en nuestra estructura de datos interna. La modificación a este algoritmo es necesaria debido a que la búsqueda del triángulo que contiene el vértice, no sólo depende de la estructura, sino también, de la versión en la cual se va a insertar el vértice. El algoritmo 1, al cual denominamos *PS-Busqueda*, realiza

este procedimiento.

Algoritmo 1: PS-Busqueda(Vértice p, EstructuraInterna T, Entero t), algoritmo propuesto por Devillers et al. (2002) y modificado para hacer uso de la estructura interna.

```
   Data: Vértice p, EstructuraInterna DT, Entero t
   Result: Arista E
 1 v := VerticeAleatorio(DT,t);
 2 E := DT-CargarAristas( v, DT , t);
 3 E := Sort(E,p) ;    // Ordenar E respecto a p en sentido antihorario.
 4 e := toma una arista ∈ E;
 5 if RightOf(X, e) then
 6     e := e.Sym endif;
 7 while true do
 8     if X = e.Org or X = e.Dest then
 9         return e;
10     else
11         whichop := 0;
12         if not RightOf(X, e.Onext) then
13             whichop += 1;
14         if not RightOf(X, e.Dprev) then
15             whichop += 2;
16         if whichop is 0 then
17             return e;
18         else
19             if whichop is 1 then
20                 e := e.Onext;
21             else
22                 if whichop is 2 then
23                     e := e.Dprev;
24                 else
25                     if dist(e.Onext, X) <dist(e.Dprev, X) then
26                         e := e.Onext;
27                     else
28                         e := e.Dprev;
29 return e
```

El algoritmo, muestra el método propuesto para buscar un vértice cercano en la triangulación persistente. Para esto, los datos de entrada son: el vértice p a insertar, la estructura de datos persistente de la triangulación de Delaunay y un valor entero t, el cual indica el tiempo en el que se realizará la operación de inserción.

En la línea 1 del algoritmo se ejecuta la función *VerticeAleatorio*, la cual retorna, de manera aleatoria, un vértice de la estructura. Este procedimiento se realiza en orden $O(1)$, debido a que los vértices están indexados en una tabla *hash*.

Una vez cargado el vértice v; entonces, en la línea 2, se recuperan todas las aristas de este vértice, las cuales se encuentran almacenadas en un árbol de búsqueda equilibrado, siempre y cuando el tiempo de creación de las aristas adyacentes estén dentro del rango del parámetro T. Este procedimiento toma un tiempo de $O(k)$, donde k es el número de aristas adyacentes a v. La línea 3 ordena las aristas de v en sentido antihorario y el ordenamiento se realiza en un tiempo $klog(k)$.

En la línea 4, se toma una arista de E y, a partir de la línea 5 en adelante, el algoritmo es el mismo que el propuesto por Brown y Faigle (1997), con la diferencia adicional, que cada vez que se requiera visitar un nuevo vértice, se debe cargar, desde la estructura interna, tanto ese vértice como sus adyacentes. Luego, el costo computacional requerido en el peor de los casos es $(O(\sqrt{n} * k * log(k))$; donde n indica la cantidad de vértices en toda la triangulación, k representa el grado máximo de la triangulación.

Finalmente, dado que en la práctica el conjunto de aristas adyacentes a un vértice v suele ser muy pequeño en relación a n, entonces $\sqrt{n} * k * logk <= C\sqrt{n}$, de lo cual se concluye que el algoritmo propuesto tiene una complejidad computacional de $O(\sqrt{n})$

Se debe tomar en cuenta que el algoritmo 1, retorna una de las aristas del triángulo que contiene a p; a continuación, es necesario encontrar los otros dos vértices del triángulo que contiene a p. Esto se logra, al encontrar la arista contigua, de tal manera que ambas aristas conforman un triángulo que contiene a p.

Con los 3 vértices que conforman el triángulo que encierra el vértice p, se procede a cargar la triangulación. Este procedimiento se realiza en la estructura interna, y carga sólo los triángulos que serán modificados; es decir, aquellos que deben ser afectados por una operación *flip*. Esto permite mantener, estrictamente, la información de los cambios que serán realizados.

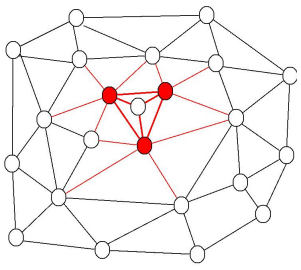


Figura 3.1.5: Carga de los triángulos que contienen al menos un vértice del triángulo que contiene el vértice a insertar.

En la Figura 3.1.5, se marcan de color rojo sólo los vértices y sus respectivas aristas, las cuales son cargadas desde la estructura interna. A continuación, se crean 3 aristas adyacentes al vértice a insertar p con los vértices del triángulo que contiene a ese vértice. Luego, se

evalúan las aristas del triángulo que contiene a p y en caso de existir alguna arista ilegal (que no cumpla con la condición de Delaunay) se debe de llevar a cabo la operación *flip* sobre esa arista, para posteriormente evaluar las 2 aristas opuestas.

En la Figura 3.1.6, las aristas de color azul, representan las aristas que serán evaluadas con el objetivo de verificar si son aristas legales o ilegales; de ser ilegales, se les aplica la operación *flip*.

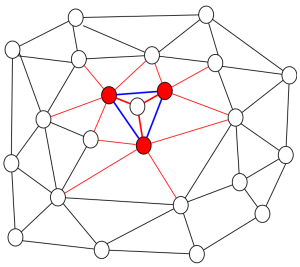


Figura 3.1.6: Aristas a ser evaluadas para realizar operaciones *flip* de color azul.

En la Figura 3.1.7, la arista de color verde representa un arista ilegal; entonces, para aplicar la operación *flip* se carga en memoria el vértice opuesto a esa arista. El nodo de color negro, en la misma Figura, representa este comportamiento.

Luego de aplicar la operación se cargan las aristas adyacentes al nodo negro, y el algoritmo continúa realizando el mismo proceso siempre y cuando se encuentre una arista ilegal.

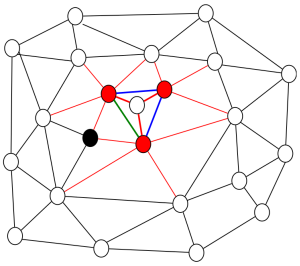


Figura 3.1.7: La arista de color verde no cumple con la condición de Delaunay; por lo tanto es ilegal, entonces el nodo de color negro será cargado para ejecutar la operación *flip*.

La Figura 3.1.8 muestra en color azul, el conjunto de aristas que serán evaluadas luego de aplicar el primer *flip*. Es importante notar que en la triangulación de Delaunay persistente sólo se carga el conjunto de vértices y aristas que serán evaluadas luego de una operación de inserción. Esto hace que la cantidad de memoria RAM esté acotada sólo por el número de vértices y aristas cargados en el proceso de evaluación.

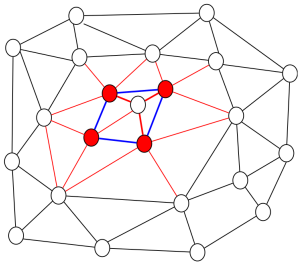


Figura 3.1.8: Aristas a ser evaluadas para realizar operación *flip* de color azul, triangulación cargada en color rojo.

El algoritmo 2, presenta la modificación al algoritmo clásico incremental para triangulaciones de Delaunay, pero para funcionar en la estructura de datos interna, la cual forma parte de la propuesta de la triangulación de Delaunay persistente.

La línea 1 del algoritmo, obtiene el identificador del árbol binario equilibrado que almacena información de las aristas del vértice p. Es importante notar que, la primera vez, este árbol está vacío. La línea 3 retorna en E una de las tres aristas que forman el triángulo que contiene a p. Las líneas $4, 5$ *y* 6 obtienen los identificadores de tres vértices del triángulo que contiene al vértice p. En las líneas 10 *a la* 18, se crean las aristas que conforman el triángulo y se insertan junto con sus vértices en los arreglos A y V respectivamente. Estás aristas serán utilizadas para analizar si se tratan de aristas legales o ilegales. El proceso de análisis se ejecuta en las líneas 19 a la 33. La parte más importante en este segmento de código, se encuentra en las líneas 25 a la 33; aquí, se evaluar directamente si una arista es ilegal, y de serlo, se cargan sólo los triángulos que se verán afectado por la operación *flip* y que posteriormente pueden desencadenar nuevas operaciones *flip*.

3.1.2.1. Prueba de Correctitud:

1. **Inicialización:** Inicialmente se tiene la triangulación de Delaunay temporal, la cual contiene todos los triángulos adyacentes al triángulo que contienen a x. Este estado se puede observar en la Figura 3.1.5. En la Figura, solo la triangulación de color rojo es cargada en una estructura de Delaunay temporal.

2. **Invarianza:** En cada iteración se mantiene una triangulación de Delaunay temporal. En cada iteración se evalúa la siguiente arista $E \in A$. Si se ejecuta una operación *flip*, entonces, se carga el par de aristas que podrían ser afectadas y puedan dar origen a nuevas operaciones *flip*. Luego, se mantiene una triangulación temporal, según el algoritmo incremental clásico, hasta que no es posible hacer más operaciones *flip*. Aquí, es importante notar que, en la práctica, el número de operaciones *flips* en una triangulación de Delaunay es constante, incluso $O(1)$ en distribuciones aleatorias de vértices (Lischinski, 1994); por lo cual, su efecto en la performance del algoritmo propuesto es mínimo.

3. **Terminación:** Dado que el número de operaciones *flips* es constante y finito, y que el algoritmo finaliza cuando no existen más aristas en el arreglo A, entonces se garantiza

que el algoritmo termina.

Algoritmo 2: DT-InsertarVértice (EstructuraInterna DT, Vértice x, entero t)

```
Data: EstructuraInterna DT, Vértice x, entero t
Result: EstructuraInterna DT con x insertado en tiempo t
1  id_x = ObtenerId(x,DT);
2  Assert( id_x ∉ DT );
3  E = PS − Busqueda(DT,x,t);
4  id_p = E.org;
5  id_q = E.dest;
6  id_r = obtener-vértice( E,p);
7  Pila A // Aristas potenciales para realizar flips
8  Pila V // Vértice en dirección opuesta a los flips
9  temp − tri = crearDT();
10 crear-Arista(DT,t,id_x,id_p);
11 crear-Arista(DT,t,id_x,id_q);
12 crear-Arista(DT,t,id_x,id_r);
13 A.insertar(id_p,id_q);
14 V.insertar(id_x);
15 A.insertar(id_q,id_r);
16 V.insertar(id_x);
17 A.insertar(id_r,id_p);
18 V.insertar(id_x);
19 for E = A.pop(), id_o = V.pop() do
20     id_ca = E.org;
21     id_cb = E.dest;
22     FaceFH = temp-tri.face( id_ca, id_cb, id_o));
23     Assert( FH != NULL);
24     id_op = FH.neighbor( FH.index( id_o ) );
25     if tmp-tri.Ilegal-Edge( E ) then
26         DT-CargarTriangulos( tmp-tri , DTI, o, t ) ;
27         tmp-tri.Flip(E);
28         DT.actualizar-tiempo(E);
29         DT.crear-Arista( id_o,id_op,t);
30         A.insertar(id_op,id_ca);
31         V.insertar(id_cb);
32         A.insertar(id_op,id_cb);
33         V.insertar(id_ca);
```

3.1.3. Carga de la Triangulación

La estructura de datos interna, no almacena información acerca de las caras, orden y orientación, así como la información de los triángulos vecinos. Esto con el objetivo de

minimizar RAM. En tal sentido, se debe realizar un procedimiento que permita cargar los triángulos, pero considerando la orientación e información correcta de la vecindad. Para esto, se hace uso del algoritmo 3. Este algoritmo recibe como entrada un vértice p y retorna todos sus vértices adyacentes, en un determinado tiempo t, ordenados en sentido antihorario.

Algoritmo 3: DT-CargarAristas(Vértice p, EstructuraInterna DT, Entero t)

```
   Data: Vértice p, EstructuraInterna DT, Entero t
   Result: Arreglo V
1  Arreglo V ;
2  id = ObtenerId(p,DT);
3  for cada nodo x en el árbol asociado a id do
4      if t < x.TE and t >= x.TC then
5          Agregar x.VD a V;
6  Sort(V,p) ;          // Ordenar V en sentido antihorario respecto a p
7  return V
```

En el algoritmo 3, la línea 2 retorna el identificador del árbol binario de búsqueda balanceado, el cual es indexado por p y está almacenado en la estructura interna. El tiempo necesario para recuperar este vértice es $O(1)$. Posteriormente, en las líneas $3, 4$ *y* 5, se obtienen todos los vértices dentro del árbol binario de búsqueda asociado a p, tomando en cuenta que, para cada nodo del árbol, el argumento t; de la función, esté dentro del rango $t_{inicial} \leq t \leq t_{final}$. Esta función permite reconstruir el conjunto de triángulos, que son necesarios, para realizar las operaciones *flip* sólo en los vértices y aristas que existen en ese tiempo t.

3.1.3.1. Prueba de Correctitud:

1. **Inicialización:** Inicialmente se tiene solo un vértice, el cual es el vértice p
2. **Invariante:** Dado $p \in T$ entonces se carga todos los vértices q adyacentes a p.

   Una vez cargados todos los vértices adyacentes a p, se ordenan en sentido antihorario a p, la correctitud de este procedimiento está dada por la correctitud del algoritmo de ordenación a utilizar.
3. **Finalización:** se garantiza que este algoritmo termina, dado que se insertan diferentes vértices, por lo tanto, sea i el número de vértices vecinos y k el número total de vértices insertados hasta el tiempo t, se garantiza que el algoritmo termina cuando $i = k + 1$.

Una vez cargada las aristas adyacentes al vértice p, es necesario regenerar el triángulo formado por estas aristas. Para lograr esto, se aplica el algoritmo 4.

Este algoritmo, en la línea 1, obtiene todas las aristas adyacentes al vértice p, posteriormente en las líneas 3 y 4 se crean los triángulos dentro de la triangulación actual t. En las líneas 5 hasta la 8, se inicializa la información de vecindad respecto a cada vértice de

cada triángulo. Esto con el objetivo que garantizar el correcto funcionamiento del algoritmo incremental.

Algoritmo 4: DT-CargarTriangulos(Triángulacion T , EstructuraInterna DT, Vértice p, Entero t)

Data: Triangulación T , EstructuraInterna DT, Vértice p, Entero t
Result: Triangulación T con triangulos adyacentes a p cargados

```
1 Arreglo V = DT-CargarAristas(p,DT,t) ;
2 Sort(V,p) ;          // Ordenar V en sentido antihorario respecto a p
3 for Cada Vértice q en V do
4     T.Crear( triángulo(p, q_Anterior, q );
5 for Cada Vértice q en V do
6     triángulo tr = T.obtenerReferencia( triángulo(p, q_Anterior, q) );
7     tr.vecino( qant ) = TrianguloSiguiente;
8     tr.vecino( q ) = TrianguloAnterior;
```

3.1.3.2. Prueba de correctitud para el algoritmo 4, el cual realiza la carga de la triangulación de Delaunay temporal, a partir de nuestra estructura interna

1. **Inicialización:** al iniciar el algoritmo, se cuenta con una sola cara, la cual consideraremos como infinita o nula (*NULL*):

2. **Invariente:** dado que $p \in T$, entonces se carga todos los vértices q adyacentes a p, este proceso se muestra en la línea 1, posteriormente se realiza la inicialización de cada cara, este proceso se observa en las líneas 5 a la 8. Es así que antes de la inserción de cada vértice, se mantiene cargada la triangulación de Delaunay correspondiente a los vértices insertados, y sus respectivos vértices adyacentes.

3. **Finalización:** sea i el número de nodos insertados hasta una iteración dada y n el número total de vértices insertados hasta el tiempo t, se garantiza que el algoritmo termina cuando $i = n+1$ en el peor de los casos.

3.1.4. Operación de Eliminación en la propuesta *walk*

En esta sección se explica cómo realizar la operación de eliminación persistente.

Para realizar la eliminación de un vértice en una triangulación de Delaunay y por lo visto en la sección 2.4, basta eliminar las aristas salientes del vértice a insertar junto con el vértice, y a continuación realizar la retriangulación del polígono resultante, como se muestra en la Figura 3.1.9 se observa en amarillo el polígono que deberá ser triangulado, siendo las aristas agregadas estar dentro del polígono.

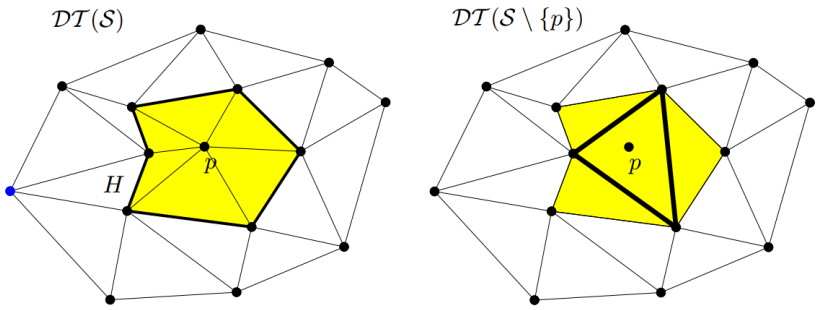


Figura 3.1.9: Eliminación de un vértice.

En el caso de la estructura de datos persistente, las aristas deberán ser actualizadas en el tiempo final como el tiempo en el cual se realiza la eliminación del vértice. y agregar las nuevas aristas con tiempo inicial igual al tiempo actual t y tiempo final infinito (aún presente en la triangulación actual).

Existen numerosos métodos para realizar la eliminación de un vértice a la triangulación de Delaunay, en tal sentido, se plantea utilizar un algoritmo de eliminación simple y eficiente de complejidad $(klogk)$ para eliminar el vértice en una triangulación de Delaunay, basada en *Shelling*, para esto se utilizara el siguiente lema (Devillers, 2002)

Lema: Considere el polígono $H = \{q_0, q_1, ..., q_{k-1}, q_k = q_0\}$ y un vértice p tal que las aristas de $q_i q_{i+1}$ pertenecen a la triangulación de Delaunay de $\{q_0, q_1, ..., q_{k-1}, p\}$. Si $|Potencia(p, circle(q_i, q_{i+1}, q_{i+2}))|$ es máxima, entonces $q_i q_{i+2}$ es una arista de la triangulación de Delaunay de $\{q_0, q_1, ..., q_{k-1}\}$ (Devillers, 2002).

Entonces la eliminación de un vértice se puede implementar mediante el mantenimiento de una estructura para almacenar *ears* y poder hallar la de menor prioridad. En esta implementación, las *ears* restantes deberán formar parte de una lista doblemente enlazada,

para representar mejor el polígono resultante en cada iteración del algoritmo 5.

Algoritmo 5: DT-Eliminar(EstructuraInterna DT , Vértice *p*,Entero *t*)

```
Data: EstructuraInterna DT, Vértice p, Entero t
Result: DT con p eliminado en tiempo t
1  Arreglo P = DT-CargarAristas( p, DT, t ) ;          // Arreglo de Vértices
2  Lista Ears ;                          // Lista doble enlazada de Índices
3  Min-Heap PQ ;
4  N = P.tamaño() ;
5  for i = 1 ; i <N ; ++i do
6      ear actual = Ear( i , ( i + 1 ) mod N, ( i + 2 ) mod N );
7      if SentidoAntihorario( actual ) then
8          PQ.insertar( Power( p , Obtener-Vértices( P , ear) ) , actual );
9
10 while PQ.vacio() == false do
11     ear actual = PQ.min();
12     if isEar( Ears, ear ) then
13         agregar( DT , ear , t ) ;      // Agregar arista a DT con tiempo de
           creación t
14         ear siguiente = Obtener-Siguiente( Ears, actual ) ;
15         ear anterior = Obtener-Anterior( Ears, acttual ) ;
16
17         PQ.insertar( Power( p , Obtener-Vértices( P, siguiente ) ) , siguiente ) );
18         PQ.insertar( Power( p , Obtener-Vértices( P, anterior ) ) , anterior ) );
19
20         Actualizar( Ears, actual) ;            // Eliminar ear y actualizar
```

3.2. Método Híbrido

Se plantea una modificación al método *Walk*, con esta modificación se acelera el proceso de búsqueda del triángulo tr que contiene a p, donde p es el vértice a insertar en la triangulación actual T; para esto se construye una estructura de localización de vértices D, que es un grafo dirigido acíclico (Kao, 1991); los nodos finales de D corresponden a los triángulos de la triangulación T actual, los nodos internos de D corresponden a los triángulos que pertenecían a la triangulación en algún tiempo anterior y que ya no forman parte de la triangulación actual.

La estructura para encontrar el vértice, fue tomada de De Berg et al. (2000) y se construye de la siguiente manera. Inicialmente se crea un triángulo lo suficientemente grande para contener dentro todos los vértices a ser insertados, esto puede observarse en la Figura 3.2.1. En esta Figura, (a) representa un único triángulo al cual denominaremos $\triangle_1$. En (b) se muestra la estructura de datos; la cual, en este caso, solo consta de un único nodo.

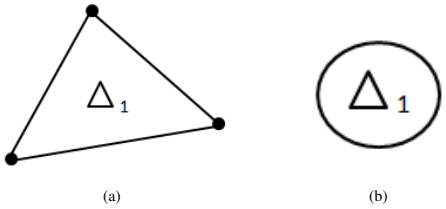


Figura 3.2.1: Estructura inicial.

De otro modo, en la Figura 3.2.2, en (a), se observa la inserción de un vértice en el interior de $\triangle_1$. Esto genera tres nuevos triángulos, a los cuales llamaremos $\triangle_2$,$\triangle_3$ y $\triangle_4$. En (b), se aprecia la estructura de datos (grafo dirigido acíclico) de la triángulacion en (a). Como se puede observar, el nodo raíz sigue siendo $\triangle_1$, es decir la triangulación en el tiempo anterior. A este nodo, se añaden los nodos $\triangle_2$, $\triangle_3$ y $\triangle_4$ que corresponden a los nuevos triángulos generados al insertar el nuevo vértice; de este manera mantenemos tanto la versión actual de la triangulación como las anteriores.

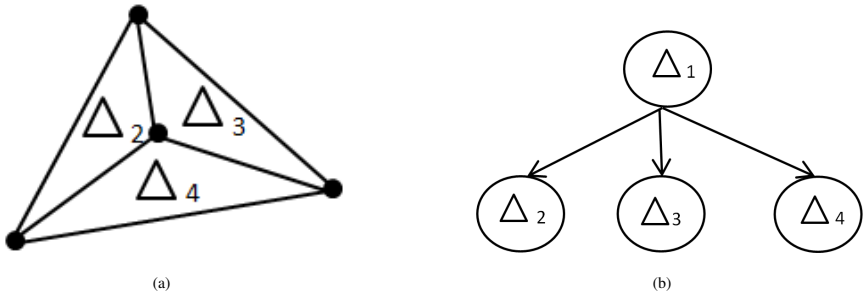


Figura 3.2.2: Estructura inicial después de insertar un vértice.

Además, para realizar una operación *flip*, en esta estructura de datos, se debe encontrar los dos triángulos que comparten la arista en la cual se llevará a cabo la operación. Por ejemplo, en la Figura 3.2.3 se muestra este comportamiento. En (a1) se aprecia que $\triangle_7$ y $\triangle_8$ tiene una arista ilegal y por lo tanto se debe realizar una operación *flip* de modo que la triangulación cambia, eliminando $\triangle_7$ y $\triangle_8$ y generando los triángulos $\triangle_{11}$ y $\triangle_{12}$, tal como se muestra en (a2).

Por otro lado, esta variación en la triangulación debe generar un cambio en la estructura de datos que la representa. Esto se puede notar en (b1) y (b2). Donde (b1) y (b2) son los grafos acíclicos dirigidos de (a1) y (a2) respectivamente. En (b2) los triángulos $\triangle_{11}$ y $\triangle_{12}$ tiene como padres a los triángulos $\triangle_7$ y $\triangle_8$, esto por que $\triangle_7$ y $\triangle_8$ fueron remplazados por $\triangle_{11}$ y $\triangle_{12}$; de esta manera todos los nodos hoja representan la triangulación actual, mientras que los demás nodos en el grafo conforman triángulos en tiempos pasados.

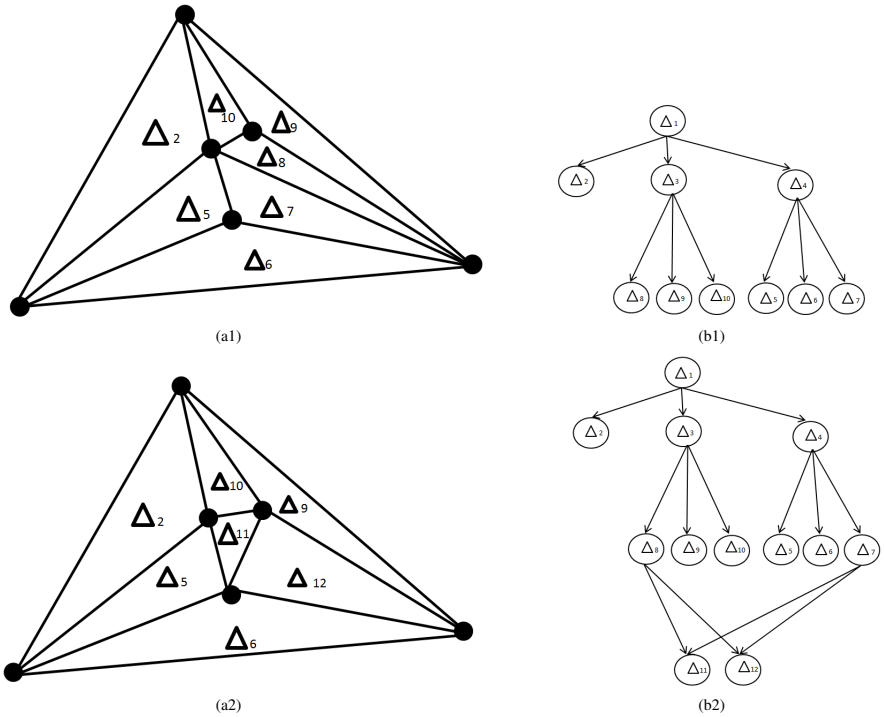


Figura 3.2.3: Estructura inicial después de insertar - *flip*.

Entonces se puede resumir la estructura de datos en los algoritmos 6 y 7.

Algoritmo 6: InsertarVértice(Estructura *D*, Vértice *x*)

Data: Estructura *D*, Vértice *x*
Result: Estructura *D* con *x* insertado

```
1  Triángulo tr = BuscarTriangulo(D.root);
2  Vértice p = tr->p0;
3  Vértice q = tr->p1;
4  Vértice r = tr->p2;
5  Triángulo newchild0 = Triángulo(p,q,x);
6  Triángulo newchild1 = Triángulo(q,r,x);
7  Triángulo newchild2 = Triángulo(r,p,x);
8  tr->childs.insertar( newchild0 );
9  tr->childs.insertar( newchild1 );
10 tr->childs.insertar( newchild2 );
```

Algoritmo 7: BuscarTriangulo(Triángulo *tr* , Vértice *x*)

Data: Triángulo *tr*, Vértice *x*
Result: Triángulo *tr* que contiene a *x*

```
1  if tr->es-Hoja() then
2      return tr;
3  for Cada Triángulo ctr en tr do
4      if ctr->contiene(x) then
5          return BuscarTriangulo(ctr,x);
```

En el algoritmo 7 en la línea 1 llegamos al caso base, en el cual hemos llegado al triángulo que contiene a x, de las líneas 3 a la 5, se busca un triángulo recursivamente (en los nodos hijos) que contenga a x.

3.3. Método Graph

La estructura de datos vista anteriormente mantiene persistencia, pues contiene todos los triángulos que alguna vez formaron parte de la triangulación, además de los triángulos de la triangulación actual como nodos finales del grafo, además si se agregara un identificador de tiempo en cada nodo de la estructura y almacenaría en un arreglo de manera secuencial ordenados de manera no descendente por el tiempo asociado, entonces se podría recuperar una triangulación en un tiempo dado t haciendo uso de búsqueda binaria sobre este arreglo y así poder recuperar los nodos finales de la estructura en ese determinado tiempo.

3.4. Persistencia Completa

En esta sección se presenta un aporte adicional. Además de la persistencia parcial, la estructura de datos soporta persistencia completa, como en la Figura 3.4.1.

Para lograr este tipo de persistencia, se añade un valor entero, a cada nodo del árbol binario balanceado de búsqueda. Este valor indica la versión a la cual pertenece una arista y, de esta forma, todas las operaciones quedan igual a la persistencia parcial.

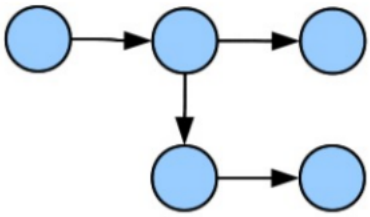


Figura 3.4.1: Persistencia Completa (Demaine et al., 2007).

Todos los algoritmos aplicados en la propuesta de persistencia parcial funcionan de manera similar en la persistencia completa, con la diferencia que ahora es necesario tomar en cuenta el valor de la versión.

Una de las dificultades, en el método de persistencia completa, es la de encontrar el vértice en el árbol de versiones desde donde iniciar el recorrido para encontrar el triángulo que contiene el vértice a insertar. Este procedimiento resulta complicado en contraste con su versión parcial.

Tomando en cuenta el árbol de versiones de la Figura 3.4.1, el problema se reduce a

encontrar un nodo aleatorio ancestro de la versión que se va a modificar, entonces, se plantea usar programación dinámica (Fischer y Heun, 2006) para encontrar el i-ésimo ancestro del nodo, es decir el i-ésimo nodo en su camino a la raíz, con esto se garantiza encontrar dicho vértice con un costo computacional de $O(logn)$ de inserción por nodo, y un costo computacional de consulta $O(logn)$ (Bender y Farach-Colton, 2000; Bender et al., 2001).

El enfoque se centra en preprocesar, mediante programación dinámica, sub-matrices de longitud 2^k. Manteniendo una matriz $M[N][logN]$ donde $M[i][j]$ es el 2^j ancestro de i.

Algoritmo 8: LCA-Calcular(Matriz M, Arreglo profundidad, Vértice p , Vértice $padre-p$)

Data: Matriz M, Arreglo profundidad,Vértice p , Vértice $padre-p$
Result: M con p insertado

```
1 id = ObtenerId( p ,DT);
2 pid = ObtenerId( padre-p , DT);
3 M[ id ][ 0 ] = pid ;
4 profundidad[id] = profundidad[pid] + 1;
5 for i = 1 ; i <log2( profundidad[id] ); ++i do
6     M[id][i] = M[ M[id][i-1] ][ i -1 ];
```

En el algoritmo 8, se calculan todos los 2^i ancestros de p. En las líneas 1 y 2 se obtienen etiquetas indexadas a través de los vértices. En la línea 3 se considera el caso base de la recursión, es decir el 2^0 ancestro de p es su padre. En la línea 5, se calcula la profundidad del nodo p, esto para obtener el i-ésimo ancestro de p; posteriormente, en las líneas 5 y 6 se calculan el 2^i ancestro de p, para $i > 0$ que define 2^i ancestro de p como 2^{i-1} ancestro del 2^{i-1} ancestro de p.

3.4.0.1. Prueba de Correctitud:

1. **Inicialización:** Al iniciar el algoritmo, el 2^0 ancestro del vértice p es el padre de p.
2. **Invariente:**

   Al final de cada iteración, se calcula el 2^i ancestro de p, de la siguiente función de recurrencia, donde T(x) representa el padre de x:

$$Ancestro(i,x) = \begin{cases} T(x), & \text{if } i = 0. \\ Ancestro(i-1, Ancestro(i-1,x)), & \text{otherwise.} \end{cases} \tag{3.1}$$

3. **Finalización:** La finalización del algoritmo está garantizado, dado que la altura del árbol de versiones tiene un valor $<= n$ donde n es el número de versiones. Por lo tanto, sea m el número de ancestros de p, se garantiza que el algoritmo termina en el peor caso cuando $i > log2(n+1)$.

Algoritmo 9: AncestroAleatorio(Matriz M, Arreglo profundidad,Vértice p)

```
   Data: Matriz M, Arreglo profundidad,Vértice p
   Result: M con p insertado
1  idVerticeAleatorio = ObtenerIndice(p);
2  nancestro = Random( profundidad[id]);
3  for i = 0 ; i <log2( profundidad[id] ); ++i do
4      if nancestro&(1 << j) then
5          idVerticeAleatorio = M[idVerticeAleatorio][i];
6  return idVerticeAleatorio;
```

En el algoritmo 9, se calcula un nodo aleatorio ancestro de p. En la línea 1, se obtiene la etiqueta de p. En la línea 2, se obtiene un número aleatorio, el cual indica el i-ésimo ancestro de p. En la línea 3 se inicializa el resultado, el cual deberá ser inicializado como p, en las líneas de 4 a 6, se descompone el i-ésimo ancestro en sumas de potencias de 2, en la línea 5, consultamos si en la representación binaria de *nancestro* el bit que corresponde a la posición i está encendido, en caso de estarlo, *idVerticeAleatorio* es actualizado como su i-ésimo ancestro.

3.4.0.2. Prueba de Correctitud:

1. **Inicialización:** Inicialmente el 0 ancestro de p es p
2. **Invariente:** Dada la representación binaria de *nancestro*, randomvertex es actualizado a ser el siguiente 2^i ancestro de p.
3. **Finalización:** Teniendo en cuenta que la altura del árbol tiene un valor $<= n$, donde n es el número de versiones, y sea m el número de ancestros de p; entonces, se garantiza que el algoritmo termina en el peor caso cuando $i > log_2(n+1)$.

Capítulo 4

Pruebas y Resultados

En el presente capítulo se explican las pruebas y resultados realizados durante la investigación. Las pruebas fueron divididas en dos grupos. El primero se enfoca en analizar la complejidad temporal y espacial de los tres métodos propuestos: *Walk*, *Híbrido* y *Graph*, en contraste con la librería CGAL. Es importante destacar aquí que los tres métodos son persistentes, mientras que *CGAL* no, por lo tanto, en este experimento se pretende demostrar que la complejidad computacional de los métodos con persistencia son similares al propio *CGAL*, quien no almacena información temporal.

En el segundo grupo de experimentos se analizó el comportamiento temporal y espacial de las estructuras persistentes, pero en un conjunto de datos aleatorios. Finalmente, se realizaron pruebas con datos aleatorios para analizar el comportamiento de la estructura de datos completamente persistente. En este último caso se hicieron pruebas con el método *Walk* debido a que es el único método que posee persistencia completa.

4.1. Grupo de experimentos en Persistencia Parcial

Para el desarrollo de los experimentos, se tomó en cuenta dos bases de datos, una conformada por un conjunto de imágenes y la otra por vértices aleatorios. A continuación se describen cada una de ellas.

El primer experimento fue desarrollado utilizando una base de datos de 17 imágenes de diferentes tamaños. El algoritmo de *Harris* (Yi-bo y Jun-jun, 2011) fue aplicado a cada una de las imágenes para obtener el conjunto de *pixels* más representativos de la imagen; a estos *pixels* se les denomina *keypoints*. Posteriormente, los *keypoints* fueron almacenados en archivos y a partir de ellos se generó la triangulación de Delaunay con los 3 métodos propuestos.

En el segundo experimento se realizaron varias pruebas con distintos conjuntos de vértices aleatorios. Se tomaron 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000, 55000 y 60000 vértices. En ambos experimentos se realizaron comparaciones

en cuanto a costo temporal y costo espacial.

A continuación se describen los resultados obtenidos para cada uno de los grupos de pruebas.

4.1.1. Experimentos en imágenes

En la Figura 4.1.1 se puede observar tres columnas, la primera muestra las imágenes reales, en la segunda columna se observa el resultado de aplicar el algoritmo de *Harris* (Yi-bo y Jun-jun, 2011) sobre las imágenes de la primera columna para obtener el conjunto de *pixels* más representativos de las imágenes (*keypoints*); finalmente, en la tercera columna se muestra la triangulación de Delaunay efectuada sobre los vértices de la segunda columna.

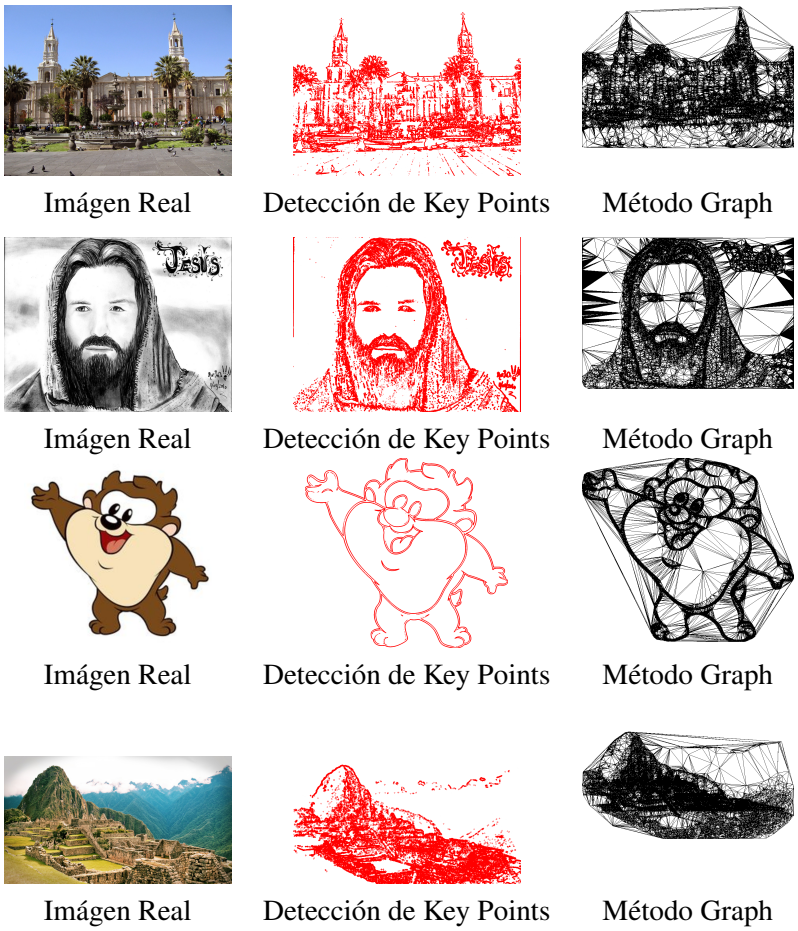


Figura 4.1.1: Resultados de aplicar triangulación.

4.1.1.1. Resultados de costo en CPU

En tabla 4.1.1 se muestra los resultados obtenidos (en segundos) al aplicar los métodos propuestos (*Walk*, *Híbrido* y *Graph*) en comparación con los resultados de *CGAL*

En la tabla, la columna 1 indica el nombre de las imágenes, la columna 2 representa el número de vértices (*keypoints*) obtenidos en cada imagen. Finalmente, las columnas 3,4 5, y 6 son los resultados, en segundos, del costo computacional de los algoritmos *Walk*, *Híbrido*, *Graph* y *CGAL*, respectivamente.

En los resultados se puede observar que, a medida que el número de vértices es mayor, el método *Graph* presenta un mejor desempeño que *CGAL*.

Por otro lado, el método *Walk* e *Híbrido* tienen un desempeño inferior, debido a que el primero hace uso del algoritmo de búsqueda modificado del propuesto por Brown y Faigle (1997) el cual incrementa su costo computacional en un factor de $O(\sqrt{n})$; mientras que en el segundo, combina la estructura de datos interna con la estructura de datos del método *Graph*, de modo que es necesario cargar los triángulos de la estructura interna cada vez que se realiza una inserción.

Archivo	Vértices	Walk	Híbrido	Graph	CGAL
img1	1139	0.318	0.320	0.160	0.066
img2	3383	1.066	1.011	0.529	0.242
img3	3805	1.947	1.159	0.571	0.364
img4	3883	1.435	1.164	0.588	0.278
img5	4048	1.337	1.202	0.585	0.308
img6	6531	2.723	2.140	1.006	0.648
img7	9000	3.465	3.634	1.461	0.874
img8	12165	5.257	3.965	2.002	1.296
img9	20767	11.382	6.784	3.702	2.621
img10	20767	11.491	6.894	3.679	2.603
img11	29495	17.547	10.813	5.994	5.147
img12	31768	22.165	10.776	6.734	5.417
img13	112780	105.710	39.867	27.152	38.415
img14	112780	109.256	40.605	25.756	39.075
img15	121279	107.771	46.189	27.097	34.907
img16	150156	144.212	52.497	33.533	44.378
img17	168093	150.886	61.744	39.408	49.405

Tabla 4.1.1: Análisis del costo temporal (en segundos) de los métodos *Walk*, *Híbrido*, *Graph* y *CGAL* en imágenes.

4.1.1.2. Resultados de costo en RAM

De la misma manera que en el experimento para evaluar el costo temporal, o costo en *CPU*, en la tabla 4.1.2 se muestran los resultados del consumo de memoria *RAM* de los tres métodos propuestos *Walk, Híbrido y Graph* comparados con *CGAL*.

En la tabla 4.1.2 se observa que, de los tres métodos propuestos para construir la estructura de datos persistente, el menor consumo de memoria *RAM* lo obtuvo el método *Graph*. Es importante notar, en este momento, que la librería *CGAL* requiere menos memoria RAM, sin embargo, *CGAL*, no es persistente, lo cual implica que no mantiene información de versiones

Archivo	Nro Puntos	Walk	Híbrido	Graph	CGAL
img1	1139	55,756	56,584	2372	448
img2	3383	58,872	61,372	6684	740
img3	3805	59,564	62,320	7488	736
img4	3883	59,572	6236	7872	740
img5	4048	59,700	62,676	7932	740
img6	6531	63,232	67,816	12,652	1032
img7	9000	66,728	73,112	17,564	1440
img8	12,165	70,860	79,676	23,360	1728
img9	20,767	83,788	98,508	40,624	2808
img10	20,767	83,788	98,508	40,624	2808
img11	29,495	95,424	117,204	56,984	3832
img12	31,768	98,864	122,144	63,244	4008
img13	112,780	212,772	292,400	215,456	13,672
img14	112,780	212,772	292,400	215,456	13,672
img15	121,279	224,564	309,768	230,840	14,628
img16	150,156	268,820	374,856	292,236	18,028
img17	168,093	293,220	410,060	324,576	20,120

Tabla 4.1.2: Análisis del costo espacial (en Kbytes) de los métodos *Walk*, *Híbrido Graph* y *CGAL* en imágenes.

pasadas y en tal sentido es de esperar que su complejidad espacial o consumo de memoria RAM sea muy inferior a los 3 métodos propuestos.

4.1.2. Experimentos en conjuntos de datos aleatorios

4.1.2.1. Resultados de costo en CPU

De la misma manera que experimentos anteriores, en la tabla 4.1.3 se puede apreciar el comportamiento temporal de los algoritmos propuestos *(Walk*, Híbrido y *Graph)* en comparación con *CGAL* para la construcción de la triangulación de Delaunay para un determinado número de vértices. Aunque *CGAL*, no admite persistencia en ninguno de sus tipos, el objetivo fue analizar si alguno de los algoritmos propuestos, además de ser persistentes, podían ejecutar la triangulación de Delaunay de manera competitiva con *CGAL*.

En la gráfica 4.1.2 se puede notar que el método *Walk*, para datos aleatorios, tiene un costo computacional mayor a los métodos *Híbrido* y *Graph* y obviamente a *CGAL*; además, podemos apreciar que el método *Graph* tiene un costo computacional inferior a los dos métodos propuestos (*Walk* e *Híbrido*) e incluso es competitivo con *CGAL*. En resumen, el método propuesto *Graph* mostró ser el más eficiente de los métodos propuestos.

Archivo	Nro Puntos	Walk	Híbrido	Graph	CGAL
10000random	10000	4.247	3.066	1.662	1.019
15000random	15000	7.922	4.686	2.646	1.886
20000random	20000	9.750	6.359	3.655	2.831
25000random	25000	13.172	8.245	4.653	3.502
30000random	30000	17.020	10.300	5.715	4.575
35000random	35000	20.447	11.808	6.754	5.655
40000random	40000	24.077	14.351	8.237	6.900
45000random	45000	29.582	14.866	8.940	9.094
50000random	50000	34.679	18.528	11.631	9.494
55000random	55000	37.447	19.756	11.962	11.732
60000random	60000	41.438	21.794	12.981	12.778

Tabla 4.1.3: Análisis del costo temporal (en segundos) de los métodos *Walk*, *Híbrido Graph* y *CGAL* en puntos aleatorios.

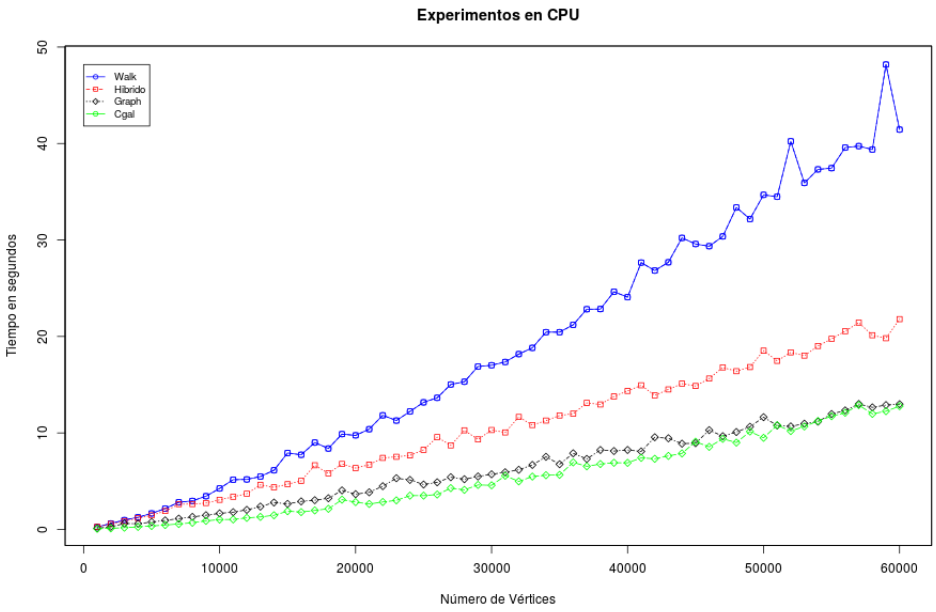


Figura 4.1.2: Comparación de desempeño en CPU entre métodos propuestos y CGAL (Diagrama de Lineas) en puntos aleatorios.

Este comportamiento se debe a que en el método *Graph*, el algoritmo para encontrar el triángulo que contiene al vértice que se pretende insertar, tiene un costo computacional $\log n$, en relación al método *Walk* cuyo algoritmo de búsqueda tiene un costo de $\sqrt{n}$. De otro lado, el método híbrido añade, a la estructura de datos de *Walk*, una estructura adicional para permitir que la búsqueda del triángulo se realice en tiempo $log(n)$, sin embargo, al requerir cargar todos los triángulos afectados por el proceso de inserción, su costo computacional se incrementa y, es por esta razón, que su desempeño es inferior al método *Graph*.

En conclusión, podemos decir, que el método *Graph* es el método con mejores resultados en cuanto a costo computacional y, cumple con el objetivo de desarrollar una estructura

de datos persistente que minimice el tiempo computacional al realizar operaciones sobre la estructura. En este caso específico, las operaciones de inserción.

4.1.2.2. Resultados de costo en RAM

En la tabla 4.1.4 se puede apreciar los resultados en el consumo de memoria RAM (Kbytes) de los algoritmos propuestos *(Walk, Híbrido y Graph)* en comparación con CGAL. Aunque CGAL, no admite persistencia en ninguno de sus tipos, el objetivo fue analizar si alguno de los algoritmos propuestos, además de ser persistentes, podían ejecutar la triangulación de Delaunay de manera competitiva con CGAL.

En la Figura 4.1.3 se aprecia el diagrama de líneas de los datos de la tabla 4.1.4, y de esta Figura podemos notar que el método *Graph* requiere menos memoria que los métodos *Híbrido* y *Walk*.

Archivo	Nro Puntos	Walk	Híbrido	Graph	CGAL
10000random	10000	68,684	76,336	20,232	1440
15000random	15000	75,680	87,028	30,628	2048
20000random	20000	83,376	98,504	40,228	2660
25000random	25000	90,232	109,196	49,600	3228
30000random	30000	97,236	120,020	61,020	3832
35000random	35000	105,508	132,300	70,788	4524
40000random	40000	112,504	142,992	80,332	5060
45000random	45000	119,500	153,816	89,836	5584
50000random	50000	126,484	164,508	99,208	6188
55000random	55000	133,492	175,200	108,580	6792
60000random	60000	140,488	185,892	122,048	7372

Tabla 4.1.4: Análisis del costo espacial (en Kbytes) de los métodos *Walk*, *Híbrido Graph* y *CGAL* en puntos aleatorios.

Se puede concluir que, sin incluir la implementación en *CGAL*, el método que consume menos memoria RAM es el método *Graph*, esto debido a que en distribuciones aleatorias la profundidad del grafo es $O(logn)$.

4.1.3. Experimentos de consultas en conjuntos de datos aleatorios

De la misma manera que experimentos anteriores, en la tabla 4.1.5 y en el digrama 4.1.4 se puede apreciar el costo temporal de los algoritmos propuestos *(Walk, Híbrido y Graph)* sobre una triangulación de 60000 vértices, donde se realizan consultas para poder recuperar una determinada triangulación en un tiempo anterior t, de un total de t vértices, para los algoritmos *(Walk*, Híbrido y *Graph)*.

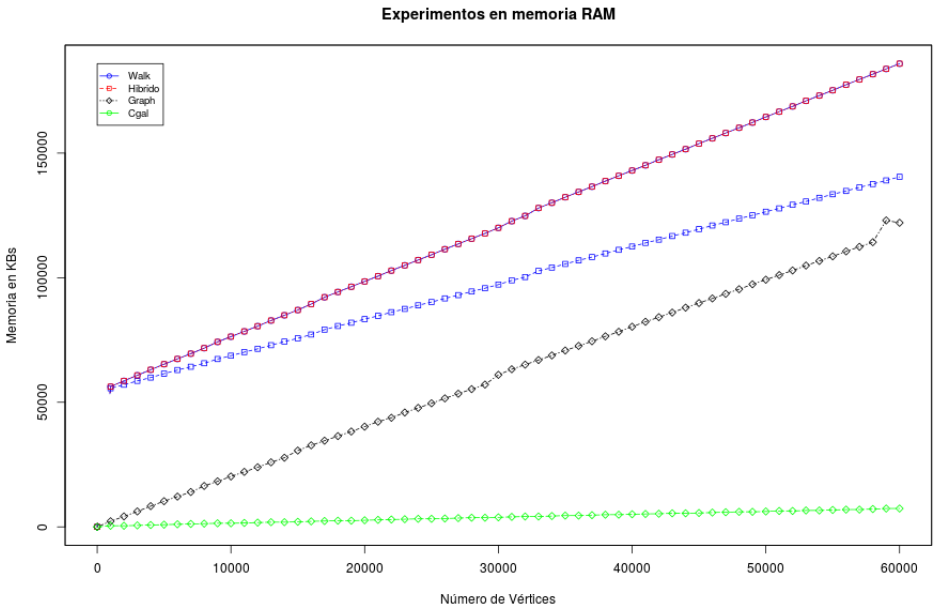


Figura 4.1.3: Comparación de desempeño en memoria RAM entre métodos propuestos y CGAL (Diagrama de Lineas).

Nro Puntos	Walk	Híbrido	Graph
10000	0.052	0.053	0.094
15000	0.054	0.054	0.133
20000	0.054	0.056	0.171
25000	0.055	0.056	0.208
30000	0.058	0.060	0.243
35000	0.058	0.061	0.279
40000	0.060	0.067	0.308
45000	0.063	0.068	0.337
50000	0.066	0.067	0.359
55000	0.068	0.068	0.380
60000	0.069	0.071	0.398

Tabla 4.1.5: Análisis del costo temporal (en segundos) al cargar una triangulación en los métodos *Walk*, *Híbrido* y *Graph* en puntos aleatorios.

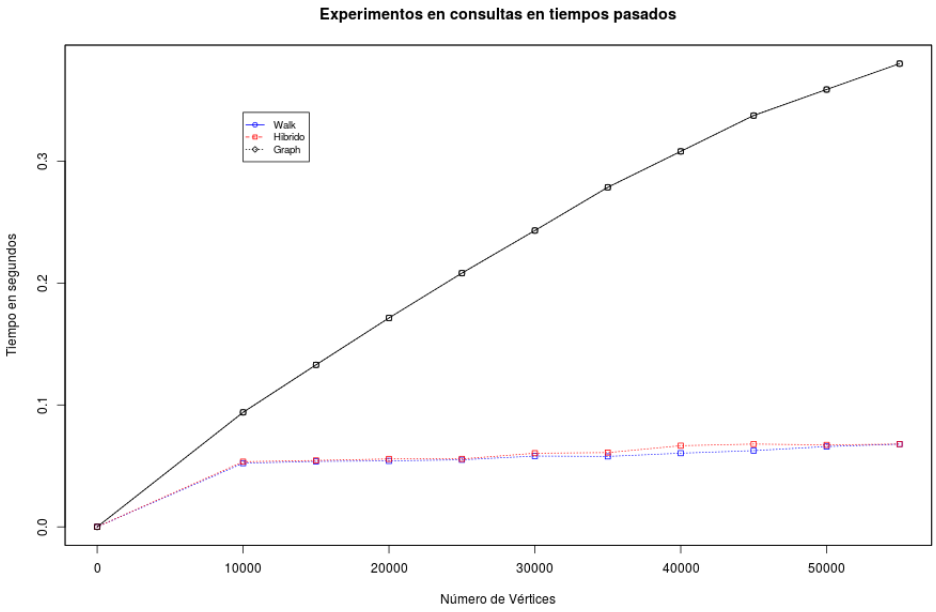


Figura 4.1.4: Consultas en tiempos anteriores para la triangulación de Delaunay.

Como en experimentos anteriores, en la tabla 4.1.6 y en la Figura 4.1.5 se puede apreciar el costo computacional de cargar completamente una triangulación en un tiempo anterior t para los métodos *Walk*, Híbrido y *Graph*

Nro Puntos	Walk	Híbrido	Graph
10000	0.316	0.314	0.271
15000	0.403	0.408	0.432
20000	0.506	0.507	0.583
25000	0.631	0.609	0.738
30000	0.713	0.713	0.876
35000	0.813	0.823	1.020
40000	0.925	0.934	1.159
45000	1.030	1.045	1.276
50000	1.141	1.150	1.422
55000	1.254	1.273	1.561
60000	1.375	1.392	1.667

Tabla 4.1.6: Análisis del costo temporal (en segundos) al realizar consultas en los métodos *Walk*, *Híbrido* y *Graph* en puntos aleatorios.

4.2. Grupo de experimentos en Persistencia Completa

A diferencia de la persistencia parcial anterior, en este experimento, se construyó la triangulación de Delaunay teniendo en cuenta que cada operación de inserción se realiza en una versión de la estructura, de modo que cada modificación producirá una nueva versión.

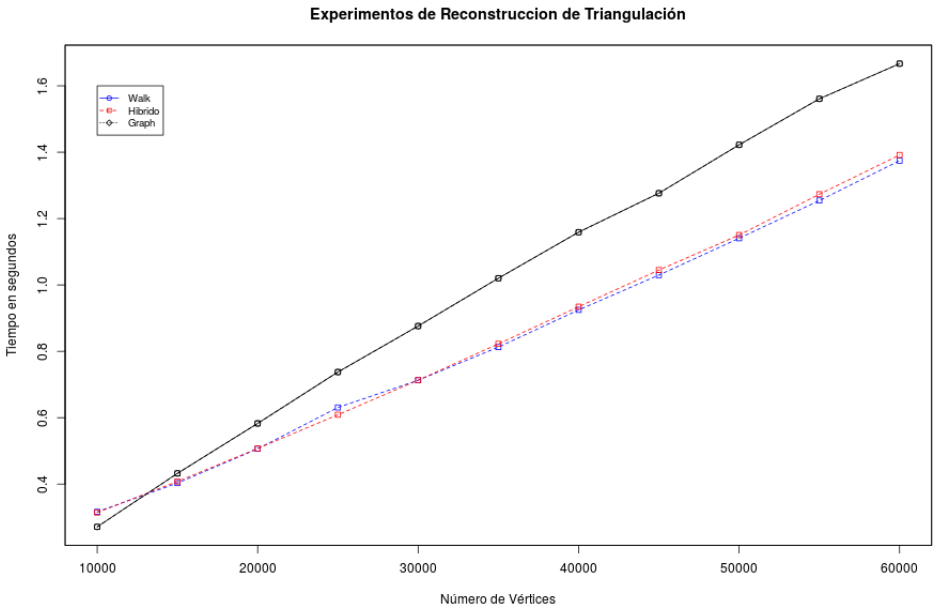


Figura 4.1.5: Carga de la triangulación de Delaunay en tiempos anteriores.

En esta sección se realizaron los experimentos para determinar el costo computacional del método de búsqueda del triángulo que contiene el vértice insertado. Se realizaron comparaciones entre el algoritmo propuesto y una técnica por fuerza bruta. Por otro lado, se realizaron comparaciones entre la estructura completamente persistente y *CGAL* modificado para simular persistencia total.

4.2.1. Evaluación del algoritmo de búsqueda

En la tabla 4.2.1 se muestran los tiempos para encontrar el triángulo que contiene el vértice a insertar, así como las operaciones de carga de los triángulos que contienen este vértice y los *flips* necesarios para construir la Triangulación de Delaunay. Se puede apreciar, en la figura 4.2.1 las diferencias en desempeño al utilizar los dos tipos de algoritmos de búsqueda del triángulo que contiene el vértice a insertar(fuerza bruta y vértice aleatorio).

Vértices insertados	Brute Force	Propuesta-Walk
10	0.009	0.005
100	0.056	0.061
1000	0.929	0.839
5000	8.045	6.732
10000	21.045	18.112

Tabla 4.2.1: Resultados de desempeño en operación de inserción (en segundos) en la triangulación de Delaunay con persistencia completa.

Para analizar el desempeño de las consultas en la estructura de datos propuesta, se creó una triangulación de Delaunay completamente persistente con 10000 vértices, luego se

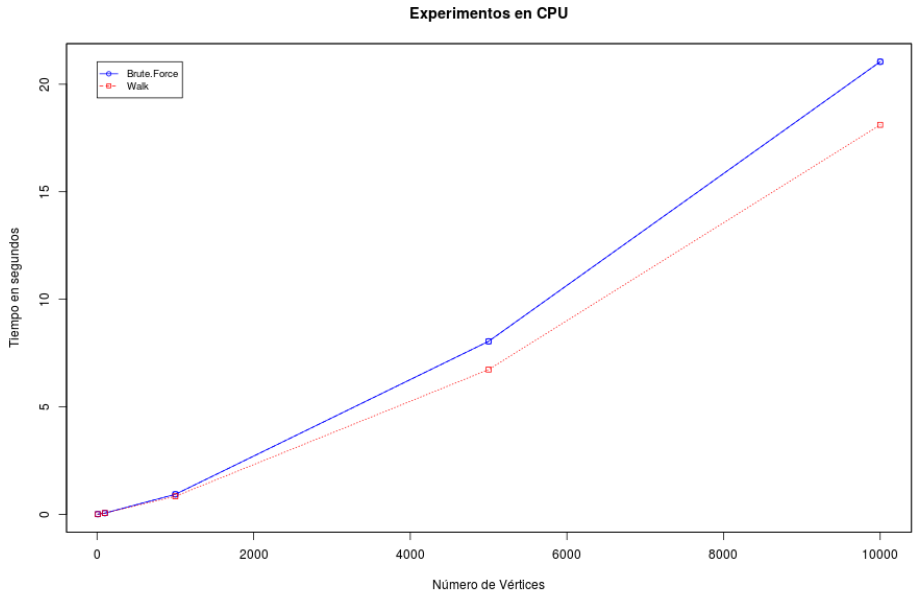


Figura 4.2.1: Comparación de los tiempos de ejecución (en segundos) de un método basado en fuerza bruta contra el método *Walk* para persistencia completa.

realizaron 10 consultas, 100 consultas y así sucesivamente, hasta 1000000 consultas y se midieron los tiempos de respuesta. Se debe notar que las consultas permiten encontrar el número de vecinos de un vértice en un determinado tiempo t, y de una determinada versión.

Nro de consultas	Tiempo
10	0.4718
100	0.4810
1000	0.4811
10000	0.4824
100000	0.4836

Tabla 4.2.2: Resultados de desempeño en operaciones de consultas en la triangulación de Delaunay (en segundos) con persistencia completa.

Como en el experimento anterior, las consultas mantienen un desempeño similar e independiente del tamaño de la triangulación del tiempo de la consulta y de la versión en la que se encuentra. En la figura 4.2.2, se puede observar que dado que los nodos están indexados directamente, el tiempo de consulta mantiene un valor casi constante por tiempo de búsqueda.

4.2.2. Evaluación de la propuesta VS CGAL

El último experimento fue desarrollado con el fin de comparar el desempeño, en cuanto a memoria RAM y eficiencia, entre la propuesta y la librería CGAL (*Computational Geometry Algorithms Library*), la cual es una librería de algoritmos geométricos ampliamente

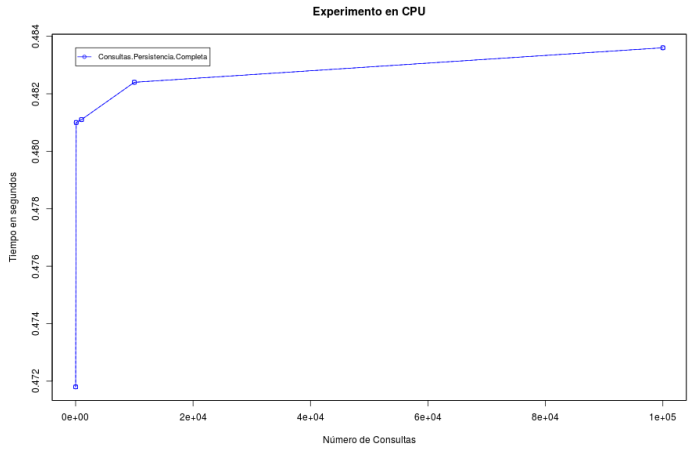


Figura 4.2.2: Tiempo de ejecución (en segundos) de operaciones de consulta para persistencia completa.

utilizada en varias aplicaciones C++ (Boissonnat et al., 2000) .

El experimento se desarrolló teniendo en cuenta que *CGAL* no posee persistencia, en tal sentido, y para ser justos, se insertaron los vértices de manera incremental tanto en *CGAL* como en la propuesta, por otro lado, luego de cada inserción, se realizó una copia de toda la estructura en *CGAL* para permitir persistencia y contrastar resultados con nuestra propuesta y, de esta forma, demostrar que, efectivamente, nuestra estructura persistente disminuye, drásticamente, la cantidad de memoria RAM y, además, es más eficiente, que mantener persistencia en *CGAL* con el método ingenuo de mantener copias.

En la Tabla 4.2.3, se puede apreciar la cantidad de memoria RAM, en Kbytes, requerida por *CGAL* y por la propuesta; también, se observan los resultados, en segundos, de la eficiencia de nuestra propuesta en contraste con CGAL.

En los resultados se puede apreciar que a medida que la cantidad de vértices a insertar se incrementa, *CGAL* consume significativamente una mayor cantidad de memoria RAM en contraste a la propuesta y, por otro lado, también, se puede notar la eficiencia computacional de nuestra propuesta persistente en contraste a la librería en cuestión.

Test	CGAL		Propuesta	
Vértices	RAM	CPU	RAM	CPU
10	16	0.020	16	0.009
100	16	0.023	16	0.051
1000	65,100	1.002	2232	0.474
5000	1,552,484	24.542	7824	2.871
10000	6,095,564	100.723	15,052	5.279

Tabla 4.2.3: Comparación de eficiencia (en segundos) y consumo de memoria RAM (en Kbytes) entre CGAL y la propuesta para persistencia completa.

En las figura 4.2.3 y 4.2.4 , se puede observar los tiempos de demora así como el uso

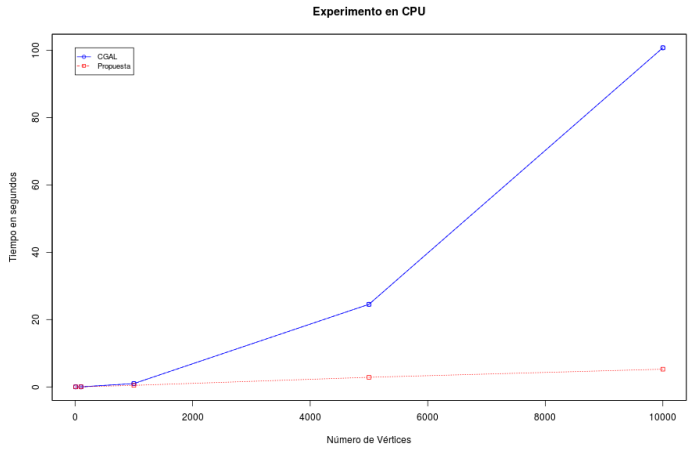


Figura 4.2.3: Comparación del tiempo de ejecución (en segundos) entre nuestra propuesta y CGAL para persistencia completa.

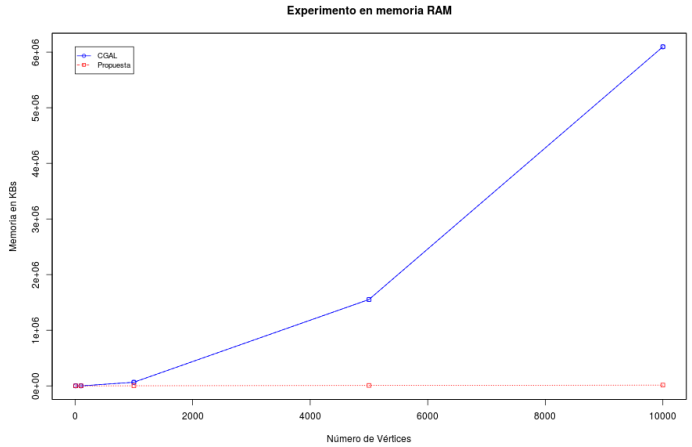


Figura 4.2.4: Comparación de memoria RAM (en Kbytes) entre nuestra propuesta y CGAL para persistencia completa.

de memoria RAM, para la propuesta dada y su implementación ingenua en CGAL, en ambos casos se confirma en la práctica, las diferencias asintóticas entre ambas implementaciones.

4.3. Análisis de complejidad de los algoritmos

En esta sección se analiza la complejidad computacional de las estructuras de datos propuestas.

La principal diferencia entre los métodos *Walk*, Híbrido y *Graph*, está dada por el procedimiento de búsqueda del triángulo que contiene el vértice a insertar en la triangulación de Delaunay, el método *Walk* elige un vértice aleatorio sobre el cual realiza un recorrido hasta encontrar un vértice del triángulo que contiene el vértice a insertar; mientras que los métodos

Híbrido y *Graph* utilizan la estructura descrita en las subsecciones 3.2 y 3.3, la diferencia entre estos dos métodos se debe a la manera en la cual se mantiene la persistencia, puesto que en el método *Graph*, solo se hace uso de una estructura de datos para almacenar todos los triángulos generados durante la triangulación, así como el tiempo correspondiente, en el cual fueron creados.

De los tres métodos propuestos, el único capaz de soportar la operación de eliminación, es el método *Walk*; esto se debe a que los otros dos métodos propuestos *Graph* e *Híbrido* aún carecen de una manera óptima de realizar esta operación.

A continuación, se evalúa cada rutina por separado.

1. Método de asignación de etiquetas (indexación): Para este método se utiliza el algoritmo descrito en la subsección 3.1.1, el cual utiliza una tabla *hash*, cuya complejidad es $O(1)$. Esta rutina se utiliza en el método *Walk* e *Híbrido*.

2. Método de carga de triángulos alrededor de un vértice: La complejidad de esta rutina es $O(klogk)$ donde k es el número de vecinos para un vértice (Cormen et al., 2001). Esta rutina se utiliza en el método *Walk*, dado que el número de triángulos adyacentes suele ser constante amortizado (Tarjan, 1985), podemos considerar que el costo de este procedimiento es de una constante C.

3. Método de búsqueda de un triángulo en una triangulación: Este algoritmo varía dependiendo del método propuesto. En el caso de *walk* tiene una complejidad computacional de $\sqrt{n}$ (Brown y Faigle, 1997), sin embargo, para los métodos *híbrido* y *Graph* su tiempo computacional es $\theta(log(n))$

La carga de triángulos depende del número de vértices vecinos de un vértice en una triangulación de Delaunay, suele ser constante amortizado (Tarjan, 1985); entonces, el método *Walk* tiene de complejidad asintótica superior de $O(C)$ al realizar la carga de los triángulos afectados por el vértice a insertar. Para el caso del *Híbrido* y *Graph*, este costo es 0, debido a que todos los triángulos ya están cargados en la estructura.

Luego, la complejidad final de la estructuras de datos persistente propuesta al insertar un vértice es de $O(\sqrt{n})$ para el método *Walk*, $O(log(n))$ para el método Híbrido y $\theta(log(n))$ para el método *Graph*.

4.4. Cuestiones de robustez y degeneraciones

La robustez del algoritmo propuesto está dada por la robustez del método de búsqueda de un triángulo en la triangulación, así como la correcta carga de la triangulación, y a las operaciones geométricas, las cuales fueron implementadas con números de punto flotante de doble precisión.

Al inicio del proceso de creación de la triangulación de Delaunay se mantiene un triángulo, el cual contendrá todos los vértices a insertar y de esta manera, se minimiza la evaluación de casos particulares; esto debido a las aristas comprendidas en la cápsula convexa envolvente a los vértices (visto en la sección 2.4.1).

4.4.1. Código y los datos

El código fuente de la estructura de datos fue desarrollado en su totalidad, utilizando el lenguaje de programación c++, sin optimización del compilador y bajo un sistema operativo de distribución Libre: Ubuntu 16.04.4.

Las pruebas fueron hechas en una computador de múltiples procesadores Intel®Core i7-5930K CPU ©3,50GHz de 8Gb de Memoria RAM, procesadores de 64 bits.

Capítulo 5

Conclusiones y Trabajos Futuros

En el presente capítulo se describen las conclusiones y trabajos futuros.

5.1. Conclusiones

En esta investigación se propusieron y desarrollaron tres nuevas estructura de datos persistentes para la triangulación de Delaunay (*Walk*, *Híbrido* y *Graph*).

El método *Walk* tiene un costo computacional, para la operación de inserción, de $O(\sqrt{n})$ y, un costo computacional para la operación de eliminación de $O(klog(k))$; donde k es el número de vértices adyacentes y n es el número total de vértices. Para el método *Híbrido*, el algoritmo de inserción tiene un complejidad de $O(log(n))$, sin embargo, no soporta eliminación. Finalmente, el método *Graph* tiene un costo de inserción de $\theta(log(n))$ y tampoco soporta eliminación.

Además, se realizaron evaluaciones de corrección de cada uno de los algoritmos propuestos y se desarrollaron pruebas con bases de datos de imágenes y vértices aleatorios, llegando a las siguientes conclusiones:

La estructura de datos, en su versión parcial, resulta competitiva respecto a otras estructuras de datos planteadas e incluso mejor que aquellas que no poseen persistencia, siendo esta última la gran diferencia. Además, puede ser usada sin hacer uso de la persistencia.

En la versión completa, la estructura de datos, muestra un avance en el estado del arte, esto debido a que, en la literatura, actualmente no existe una estructuras de datos para la triangulación de Delaunay con persistencia completa.

En la versión parcial, el método *Graph* demostró una mejor performance en relación a los métodos *Walk* e *Híbrido*, tanto en costo computacional como en uso de memoria RAM. Sin embargo *Graph* no presenta eliminación persistente. En comparación, el método *Walk*, aunque es el menos eficiente de los otros dos, posee eliminación persistente y, ya que la

eficiente del algoritmo de inserción está relacionado con la performance del algoritmo de búsqueda del triángulo que contiene al vértice a insertar, el cual es O($\sqrt{n}$), entonces es posible crear un algoritmo persistente de búsqueda más eficiente para mejorar los resultados de este método.

5.2. Limitaciones

- En la versión completamente persistente la operación de carga de aristas presenta aún inconvenientes respecto a su performance. Esto se debe a la necesidad de almacenar información acerca del número de versión y valores de tiempo de inicio y tiempo final en los cuales existen aristas adyacentes con otros vértices.
- Realizar la búsqueda del triángulo que contiene al vértice que se va a insertar tiene una complejidad amortizada de $O(\sqrt{n})$, por lo tanto, queda aún pendiente la posibilidad de minimizar esta cota mediante un algoritmo de búsqueda más eficiente.

5.3. Trabajos futuros

- Como trabajos futuros se pretende utilizar una estructura de datos espacial con el objetivo de mejorar el algoritmo de búsqueda del triángulo donde se debe insertar un nuevo vértice, siendo este el de mayor costo computacional en la propuesta *Walk*.
- Es factible minimizar la complejidad algorítmica del algoritmo de ordenación en sentido antihorario a un vértice fijo y el algoritmo del punto más cercano, haciendo uso de las nuevas tecnologías de hardware, es decir mediante programación en *GPU*.
- Se puede extender nuestra propuesta a una triangulación de Delaunay persistente en 3 o más dimensiones y, además, utilizar estructuras de datos sucintas para reducir la utilización de memoria RAM.

Bibliografía

Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405.

Avis, D. y Bremner, D. (1995). How good are convex hull algorithms? In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 20–28. ACM.

Bender, M. A. y Farach-Colton, M. (2000). The lca problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer.

Bender, M. A., Farach-Colton, M., et al. (2001). Least common ancestors in trees and directed acyclic graphs.

Bernd Gärtner, Michael Hoffmann, E. W. (2013). Computational Geometry (252-1425-00L) HS13.

Blelloch, G., Burch, H., et al. (2001). Persistent triangulations. *Journal of Functional Programming*, 11(05):441–466.

Boissonnat, J.-D., Devillers, O., et al. (2000). Triangulations in cgal. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 11–18. ACM.

Böszörményi, L. y Weich, C. (1996). Persistent data structures. In *Programming in Modula-3*, pages 349–370. Springer.

Brown, P. J. y Faigle, C. T. (1997). A robust efficient algorithm for point location in triangulations. Technical report, Tech. rep., Cambridge University.

Buchsbaum, A. L. y Tarjan, R. E. (1995). Confluently persistent deques via data-structural bootstrapping. *Journal of Algorithms*, 18(3):513–547.

Chew, L. P. (1989). Constrained delaunay triangulations. *Algorithmica*, 4(1-4):97–108.

Cignoni, P., Montani, C., et al. (1998). Dewall: A fast divide and conquer delaunay triangulation algorithm in e d. *Computer-Aided Design*, 30(5):333–341.

Cormen, T. H., Leiserson, C. E., et al. (2001). *Introduction to algorithms*, volume 2. MIT press Cambridge.

De Berg, M., Van Kreveld, M., et al. (2000). *Computational geometry*. Springer.

Demaine, E. D., Harmon, D., et al. (2007). Dynamic optimality-almost. *SIAM Journal on Computing*, 37(1):240–251.

Demaine, E. D., Langerman, S., et al. (2008). *Confluently Persistent Tries for Efficient Version Control*, pages 160–172. Springer Berlin Heidelberg, Berlin, Heidelberg.

Devillers, O. (2002). On deletion in delaunay triangulations. *International Journal of Computational Geometry & Applications*, 12(03):193–205.

Devillers, O., Pion, S., et al. (2002). Walking in a triangulation. *International Journal of Foundations of Computer Science*, 13(02):181–199.

Dietz, P. F. (1989). Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, pages 67–74. Springer.

Dietzfelbinger, M., Karlin, A., et al. (1994). Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761.

Driscoll, J. R., Sarnak, N., et al. (1986). Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM.

Fiat, A. y Kaplan, H. (2001). Making data structures confluently persistent. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 537–546. Society for Industrial and Applied Mathematics.

Fischer, J. y Heun, V. (2006). Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Annual Symposium on Combinatorial Pattern Matching*, pages 36–48. Springer.

Fortune, S. (1992). Voronoi diagrams and delaunay triangulations. *Computing in Euclidean geometry*, 1:193–233.

Kao, T. (1991). Dynamic maintenance of delaunay triangulations. *work*, 9:32.

Kaplan, H., Okasaki, C., et al. (2000). Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977.

Lee, D.-T. y Schachter, B. J. (1980). Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242.

Lischinski, D. (1994). Incremental delaunay triangulation. *Graphics gems IV*, pages 47–59.

Mehta, D. P. y Sahni, S. (2004). *Handbook of data structures and applications*. CRC Press.

Mirzaian, A. (1988). Triangulating simple polygons: Pseudo-triangulations. Technical report, Citeseer.

Overmars, M. H. (1981). Searching in the past ii: general transforms. Technical report, Tech. Rep. RUU.

Preparata, F. P. y Shamos, M. (2012). *Computational geometry: an introduction*. Springer Science & Business Media.

Ramaswami, S. (1993). Convex hulls: Complexity and applications (a survey).

Razafindrazaka, F. H. (2009). *Delaunay triangulation algorithm and application to terrain generation*. PhD thesis, United Nations University.

Roussopoulos, N., Kelley, S., et al. (1995). Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM.

Sarnak, N. y Tarjan, R. E. (1986). Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679.

Seidel, R. (1986). Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 404–413. ACM.

Shamos, M. I. y Hoey, D. (1975). Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 151–162. IEEE.

Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318.

van Emde Boas, P., Kaas, R., et al. (1976). Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127.

Yi-bo, L. y Jun-jun, L. (2011). Harris corner detection algorithm based on improved contourlet transform. *Procedia Engineering*, 15:2239 – 2243. {CEIS} 2011.